

A GIS DOMAIN FRAMEWORK UTILIZING JAR LIBRARIES AS  
COMPONENTS

A THESIS SUBMITTED TO  
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES  
OF  
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EBRU ÖZDOĞRU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF MASTER OF SCIENCE  
IN  
COMPUTER ENGINEERING

MAY 2005

Approval of the Graduate School of Natural and Applied Sciences.

---

Prof. Dr. Canan Özgen  
Director

I certified that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

---

Prof. Dr. Ayşe Kiper  
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

---

Assoc. Prof. Dr. Ali Hikmet Doğru  
Supervisor

Examining Committee Members

Assoc.Prof. Dr. Volkan Atalay (CENG, METU) \_\_\_\_\_

Assoc.Prof. Dr. Ali Doğru (CENG, METU) \_\_\_\_\_

Assoc.Prof. Dr. Onur Demirörs (Informatics, METU)\_\_\_\_\_

Assoc.Prof. Dr. Nihan K.Çiçekli (CENG, METU) \_\_\_\_\_

Dr. Meltem Turhan Yöndem (CENG, METU) \_\_\_\_\_

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

Name, Last name : Ebru Özdoğru

Signature :

# **ABSTRACT**

## **A GIS DOMAIN FRAMEWORK UTILIZING JAR LIBRARIES AS COMPONENTS**

Özdoğan, Ebru

M. S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Hikmet Doğu

May 2005, 133 pages

A Component Oriented Software Engineering (COSE) modeling environment is enhanced with the capability to import executable components and deliver applications through their composition. For this purpose, an interface layer that utilizes JAR libraries as components has been developed. Also, Domain Engineering process has been applied to Geographical Information Systems (GIS) domain and utilized towards converting the environment to a development framework. The interface layer imports JAR libraries into the COSECASE tool, which is a graphical tool supporting COSE approach and COSE Modeling Language (COSEML). As a result, systems can be designed using abstractions and then implemented by corresponding deployed components. Imported code is made available to the COSECASE environment through this interface layer. Also, Domain Analysis, Domain Design, and Domain Implementation phases of Domain Engineering process have been applied to the GIS domain. Components developed in this Domain Implementation phase have been imported into COSECASE. A simple GIS application has been designed and generated through the interface layer of COSECASE for demonstration purposes.

**Keywords:** Component Oriented Software Engineering, Component Oriented, Component Based Development, Software Component, Domain Engineering, Application Engineering, Java Bean Components.

# ÖZ

## JAR KÜTÜPHANELERİNİ BİLEŞEN OLARAK KULLANAN BİR CBS ALAN ÇERÇEVESİ

Özdoğan, Ebru

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Danışman: Doç. Dr. Ali Hikmet Doğru

May 2005, 133 sayfa

Bir Bileşen Yönelimli Yazılım Mühendisliği (BYYM) modelleme ortamına, çalıştırılabilir bileşenleri dışarıdan aktarma ve bu bileşenlerin birleştirilmesi yolu ile uygulama geliştirilme yeteneği eklendi. Bu amaçla, JAR kütüphanelerini bileşen olarak kullanmayı sağlayan bir arayüz katmanı geliştirildi. Ayrıca, Coğrafi Bilgi Sistemleri (CBS) alanı üzerinde Alan Mühendisliği süreci uygulandı ve böylelikle modelleme ortamından bir geliştirme çerçevesi olarak yararlanıldı. Arayüz katmanı, JAR kütüphanelerinin BYYM yaklaşımı ve Bileşen Yönelimli Yazılım Mühendisliği Modelleme Dili (COSEML)'ni destekleyen grafiksel bir editor olan COSECASE aracı içerisinde kullanılması sağlamaktadır. Sonuç olarak, sistemler, soyut bileşenler ile tasarlanır ve sonrasında soyutlamalara karşılık gelen var olan bileşen kodları ile gerçekleştirilir. Bileşen kodları, COSECASE'te bu arayüz katmanı sayesinde kullanılabilir. Ek olarak, Alan Mühendisliği sürecinin, Alan Analizi, Alan Tasarımı ve Alan Uygulaması evreleri CBS alanına uygulandı. Alan Uygulaması evresinde geliştirilmiş bileşenler COSECASE'e kullanılmak üzere yüklendi. Gösterim amaçlı, basit bir CBS uygulaması, COSECASE'in arayüz katmanı kullanılarak tasarlandı ve kod üretildi.

Anahtar Kelimeler: Bileşen Yönelimli Yazılım Mühendisliği, Bileşen Yönelim, Bileşen Yönelimli Uygulama Geliştirme, Yazılım Bileşeni, Alan Mühendisliği, Uygulama Mühendisliği, Java Bean Bileşenleri.

To My Minoş  
For being always with me...

## **ACKNOWLEDGEMENTS**

I would like to thank my supervisor, Assoc. Prof. Dr. Ali Hikmet Doğru, for his guidance and encouragement throughout the research. To my family, I offer sincere thanks for their emotional support.

# TABLE OF CONTENTS

ABSTRACT .....	iv
ÖZ.....	v
ACKNOWLEDGEMENTS.....	vii
TABLE OF CONTENTS .....	viii
LIST OF TABLES.....	x
LIST OF FIGURES .....	xi
CHAPTER 1 .....	1
1. INTRODUCTION.....	1
1.1. Reuse in Software Development .....	2
1.2. Motivation for Domain Engineering.....	3
1.3. Motivation for using COSEML and COSECASE Tool .....	4
1.4. Organization of the Thesis.....	4
CHAPTER 2 .....	6
2. BACKGROUND.....	6
2.1. Software Components.....	6
2.2. Software Reuse .....	8
2.3. Component Based Software Engineering (CBSE) .....	9
2.4. Domain Engineering (DE).....	10
2.4.1. Domain Analysis .....	12
2.4.2. Domain Design.....	17
2.4.3. Domain Implementation.....	18
2.4.4. Application Engineering.....	19
2.5. Java Archive (JAR) Library Files.....	20
2.6. Java Beans Technology .....	21



2.7. Bean Development Kit (BDK) and BeanBox Testing Application.....	26
2.8. COSE Modeling Language (COSEML) and COSECASE.....	27
CHAPTER 3 .....	31
3. APPLICATION OF DOMAIN ENGINEERING PROCESS TO GEOGRAPHICAL INFORMATION SYSTEMS (GIS) DOMAIN AND DOMAIN FRAMEWORK.....	31
3.1. Domain Analysis for GIS Domain.....	32
3.2. Domain Design for GIS Framework.....	36
3.3. Components for GIS Framework Domain Implementation.....	41
CHAPTER 4 .....	58
4. INTERFACE LAYER TO UTILIZE JAR LIBRARIES AS COMPONENTS	58
4.1. Importing Beans into COSECASE Tool and System Design.....	58
4.2. Experimental Results.....	71
4.3. Evaluation of Other Related Work .....	74
CHAPTER 5 .....	76
5. CONCLUSION .....	76
5.1. Work Conducted.....	76
5.2. Comments.....	77
5.3. Future Work.....	78
REFERENCES .....	79
APPENDIX A.....	83
GIS FRAMEWORK DESIGN .....	83
APPENDIX B.....	103
DETAILED DEFINITION OF DEVELOPED BEAN CLASSES IN GIS DOMAIN FRAMEWORK .....	103

## LIST OF TABLES

### TABLE

1. COSEML symbols and their meanings.....	29
2. Results of the experiments by using the interface layer.....	72
3. Results of the experiments by coding the application.....	72
4. Comparison of the values of interface layer / coding manually.....	73

# LIST OF FIGURES

## FIGURES

5. Domain Analysis phase in Domain Engineering process .....	13
6. Domain Analysis activity and sub-activities.....	15
7. Feature Model in FODA .....	17
8. Domain Design phase in Domain Engineering process .....	18
9. Domain Implementation phase in Domain Engineering process .....	19
10. Software Development Based on Domain Engineering.....	20
11. Event Handling in Java .....	22
12. Java Beans and JAR Packages .....	25
13. Bean Development Kit (BDK).....	27
14. Graphical symbols in COSEML .....	28
15. A Feature Model for the GIS Domain .....	35
16. Using a component library within the GIS Domain Framework .....	36
17. Setting configuration file in GIS framework .....	38
18. GIS Domain Framework design .....	39
19. Communication between the BeanMap and the BeanTool components in the GIS Domain Framework .....	47
20. Communication between the BeanMap and the BeanLayerVector components in the GIS Domain Framework .....	48
21. Communication between the BeanMap and the BeanLayerRaster components in the GIS Domain Framework .....	49
22. Communication between the BeanMap and the BeanLayerAnalysis components in the GIS Domain Framework .....	50
23. Communication between the BeanMap and the BeanLayerSymbolsFont components in the GIS Domain Framework .....	51
24. Communication between the BeanMap and the BeanLayerSymbolsGeometry components in the GIS Domain Framework .....	52

25. Communication between the BeanMap and the BeanLayerSymbolsDrawing components in the GIS Domain Framework .....	53
26. Communication between the BeanMap and the BeanLayerSymbolsImage components in the GIS Domain Framework .....	54
27. Communication between the BeanMap and the BeanCoordinateConverter components in the GIS Domain Framework .....	55
28. An example of using configuration file with the BeanMap component .....	56
29. Component and Interface Symbols in COSEML .....	59
30. Conceptual Layers in the Framework .....	59
31. Importing existing bean components into COSECASE .....	60
32. Using JAR files as components in the COSECASE .....	62
33. Icons of Imported Bean Component in COSECASE .....	63
34. Bean Components from JAR libraries in COSECASE .....	64
35. Beans in COSECASE .....	65
36. Properties Dialog Box in COSECASE .....	68
37. Parameters Dialog Box in COSECASE .....	69
38. Beans Menu in COSECASE .....	71

# **CHAPTER 1**

## **INTRODUCTION**

The goal of this thesis is to provide the ability to compose applications out of existing software components. This ability is facilitated by enhancing a modeling tool. The graphical models created in the COSECASE environment accommodated abstractions as well as the representation of executable components but the tool did not contain such components.

This work resulted in the capability to import existing code that can be treated as a set of complying components. A key concept in the composition of components towards an executable system is the interfaces. These structures assume the task of connecting components. However, the logical model offered by COSECASE needs compatibility with executable components for the sake of consistency and operability. The main effort in the implementation related part of this thesis was exerted for the automatic construction of COSEML compatible interfaces. Once imported code is presented in the modeling screens as components and interfaces, system integration can be handled by semi-automatic connections among the methods and events of the components.

As the application field, the Geographical Information Systems (GIS) was selected. Repeated efforts in production of such systems in the defence industry were noticed and familiarity with this field helped in this selection.

Component libraries make more sense in a restricted application domain. Naturally, Domain orientation provides the context for the efficient use of a set of components whose functionality and meaning is more specific, in a domain. Applying Domain Engineering (DE) was an intermediate step towards providing a functional environment: the GIS Domain was modeled and supporting elements were developed for the framework infrastructure. This preliminary work included a domain model and a set of components.

All the mentioned approaches are actually manifestations of the reuse concept in software engineering. This is a concept that eases many difficulties related with developing new code.

## **1.1. Reuse in Software Development**

Complexity of software systems has increased extremely during the last decade. As software systems become more complicated, they tend to be rather error prone. Many projects take many years and cost millions of dollars. Furthermore, significant portions of the system are still under development with none of the original developers. Technical knowledge is partitioned among individuals that address the scope of the project. The software community quite unanimously sees that the reuse of existing software assets is the key to overcome these problems. A high degree of software reuse offers possibilities for reducing development efforts and improving software quality. Meanwhile, there exist various methods and tools to support the development and management of very large and complex software systems.

Several approaches have been introduced to enable different kinds of reuse, e.g. source code in the form of modules, functions, classes, or components and other artifacts related to analysis, design, and architectures. The reuse of architecture artifacts is closely connected with the application of patterns [1]. In the 1970s, modules were introduced as reusable software entities. However, those modules have to be adapted by editing the source code [2]. In the 1980s, object-oriented programming introduced the concept of “class” as a basic unit of reuse [3] [12]. Inheritance associated with object-orientation is a powerful mechanism to adapt code, but object-orientation has failed to support high reuse levels. Abstraction, which has become increasingly popular through object-oriented development methodologies, is one way to deal with the complexities of large system development. Late 1990s have been the prime time for software components and component-based software engineering was introduced. The problem with components is that the bigger the component, the more specialized it tends to be. On the other hand, small components are not efficient in terms of reuse because of the development overhead. It is easier to write a specialized component for each software product.

The term Commercial Off-The-Shelf (COTS) components means reusable components sold in the marketplace. Customers and system contexts for such COTS components are not known a priori and may differ very much from each other.

Reuse expands the development context from a single project to multiple projects. Therefore, domain engineering is very popular in terms of reusability considering this point. Areas organized around classes of systems or parts of systems are called domains [4]. Product families are scoped based on commonalities between the products in an organization. A software product line is a “group of products” in a specific problem domain. For similar software products, software development based on product lines is connected with expectations for enhancements in reusability, adaptability, flexibility, and control of complexity and performance of software. These products can be developed from a common set of assets. Domain Engineering is the iterative process of definition, design, and development of related set of systems in a domain. It uses the principle of abstraction to reduce complexity [4]. It can provide the essential continuity across sub-systems and time, and concentrates on providing reusable solutions for families of systems. In the range of a specific problem domain, software product lines are derived from predefined architectures consisting of common and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application [1].

Systematically grouping reusable software components stored in a software repository is called classification. This process requires combining the knowledge of the components in the repository with the knowledge about the application domain where these components are going to be used. Common characteristics of the components are grouped and organized into a structure. The repository can easily understand this structure. Although classification methods have been researched for a long time, there are no accepted standards for classifying components [5] [6]. Therefore, the key difficulty in classification is organizing the overall repository.

Many projects of a military company include a GIS part to show activities in the tactical area. Not to develop GIS applications in every project from scratch, systematic reuse of components stored in a component library that is commercially available is needed in the organization. In this study, we will show how reusable components in the form of Java Archive (JAR) libraries can be made run-time components and used to develop applications easier.

## **1.2. Motivation for Domain Engineering**

GIS functions constitute an important element of military applications. Maps are displayed, zoom operations are done etc. A lot of effort is spent to develop GIS

applications in such projects from scratch. Projects in the organization are developed on both Unix and Windows platforms and a discipline is needed to develop GIS applications easier, in shorter time, and at a lower cost. In this study, GIS is selected as the domain and analysis phase of domain engineering is applied to the GIS domain. Experts in this organization collected their requirements. A component library, “Map-Objects Java (MOJ)” commercially available in the market, was found useful for most of the projects. A more flexible way is needed. This library has existing code as JAR archive files. To build GIS applications easier, an interface layer is developed to utilize JAR archive files as components.

### **1.3. Motivation for using COSEML and COSECASE Tool**

COSECASE is used to design a system with component-oriented approach. This tool provides graphical modeling of systems with logical and physical component representations. Components and connectors are selected by using shapes on the toolbar. Properties, methods, and events are entered into component interfaces manually. Lacking of executable components in COSECASE tool leads to developing an interface layer to utilize JAR libraries as components. JAR libraries including Java Beans that are deployed by companies are used. Beans in these libraries are shown as components on the toolbar. By such a way, existing component codes can be used in designing system. Properties, methods, in and out events of these existing components are shown as the interface of the component. In addition, applications can be generated and executed through the tool.

### **1.4. Organization of the Thesis**

The focus of the thesis is to develop an interface layer to utilize JAR libraries as components and using this interface layer to design systems with existing components. Code generation for the application and execution of the application is enabled within COSECASE that is based on COSE approach. Although this layer is used to import all JAR libraries that include Java Beans, a simple Domain Engineering process has been applied to GIS domain to develop JAR libraries that include Java Beans. This interface layer is used in Application Engineering, which uses assets from Domain Engineering process. Beyond this introductory chapter, the thesis is organized as follows: In Chapter 2, necessary background on software reusability and components, component-based methodology, domain engineering, Java Beans and JAR libraries are included. Chapter 3 describes the Domain Engineering process applied to the GIS domain.



Domain Analysis is conducted with a feature model of the domain as suggested by the Feature Oriented Domain Analysis (FODA) methodology. Domain Design and architecture of the domain is presented. Components developed in Domain Implementation phase and domain framework are explained. In chapter 4, interface layer that is developed to utilize JAR libraries as components for COSECASE is presented. This interface layer is used in Application engineering process to generate applications in the domain. Chapter 5 concludes, and presents work that has been conducted in the thesis and further work. Framework design is modeled with Unified Modeling Language (UML). Documentation for the design is represented in Appendix A. Components and framework developed in Domain Implementation phase are given in detail in Appendix B.

## CHAPTER 2

### BACKGROUND

#### 2.1. Software Components

Software components in software are similar with Integrated Circuits (ICs) in electronics. They are "black boxes" that encapsulate functionality and provide services based on a protocol. The protocol is a contract between the developer of the component and its user. It defines that how the component will behave in various circumstances.

There are so many definitions about software components in the literature:

*A software component is any standard, reusable, and previously implemented unit that has a function in a well-defined architecture and can be replaceable [7].*

*A component is encapsulated, distributable, and executable piece of software that provides and receives services through well-defined interfaces [9].*

*Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system [4].*

All the definitions above are based on the reusability of software components. Software components provide specific functionality that can be reused in different places. Any software component conforming to the same protocol can be replaced with another one performing the same function. Programmers can then connect software components together to create different applications, just as electrical engineer wires together components to create an electronical device. Writing applications with software components instead of writing a traditional program is analogous to wiring together ICs to build an electronical device instead of using discrete components or rolling our own inductors. Software reusability has long been a holy grail of software engineering and a major goal of object-oriented programming. Software components are designed to be reusable, even interchangeable. Since,

components are previously implemented and tested, the system becomes more reliable.

Software components hide implementation, conform to interfaces, and encapsulate data, just like classes do in object-oriented languages. Almost all software components are also classes, but their difference from classes is that components conform to a defined software component protocol.

Components interact with each other by using their interfaces. An interface is a set of operations, which can be invoked by other components. Understandable specifications of the involved syntactic and behavioral interfaces are needed. A component framework supports components conforming to certain standards, and regulates the interaction between components. A well-defined conceptual framework is required as a reliable foundation.

Although components are typically viewed as corresponding to a compilation unit, their level of granularity may distinguish them. Architectural level components correspond to subsystems or other independent units. At the code level, when compared to objects, components are larger, targeted more towards the user than the developer, conform to industry-standard interfaces, and are distributed in binary rather than source code form. Another key attribute that helps to distinguish components from other development artifacts is that they are separable from their original context. Thus, components are usable in other contexts.

The correlation of component concepts with marketplace concepts defines the term “Commercial Off-The-Shelf” (COTS). If components are to be sold as separate products, they need to be generalized to enable reuse in different contexts. “Components are viable only if the investment in their creation is returned as a result of their deployment” [4]. The cost-effectiveness of component production depends on the reusability of the produced software components. Generalization in the usage of components brings some inconsistencies. There is an inherent mismatch between the ability to easily replace components and the interests of COTS software vendors [10]. COTS components should be reusable in as many contexts as possible, and the adaptation costs for customers should be low. COTS products are very important for large system development. These systems are created by integration of COTS products from different sources. COTS-based system development is an act of composition and is shaped by realities of the COTS marketplace [11] [12]. This strategy considers

system definitions and marketplace simultaneously. On the other hand, traditional system development firstly identifies requirements, and then defines architecture, and then implements the system. Therefore organizations must learn COTS-based system development to create large systems.

## **2.2. Software Reuse**

Software Reuse is the practice of using existing software components –building blocks- to develop new applications. The idea of developing software once and using it for a variety of different requirements has been a driving force of software engineering methods for a long time. It is easier to reuse the artifacts than it is to develop software from scratch. To develop software by using existing components, it must be known what the components do. Reusable software elements can be code segments, executable programs, requirements, knowledge, design and architectures, test data and test plans, or software tools.

Software reuse is the practice of developing new applications from existing software. This offers the potential to reduce the time, cost and effort needed to develop and maintain high-quality software [5]. It plays a vital role in enhancing productivity, maintainability, portability, quality, and standards of software products.

The two primary techniques with software reusability are the compositional and generative techniques. Compositional techniques relate to creation of new software from existing components retrieved from a reusability library [13]. In the generative technique, reusable components are built into a tool. Much of the work of selecting, customizing, and composing the components is automated [6]. The generative techniques are an extension of the compositional approach. Compositional techniques are more fundamental, since basic components must be well understood to be incorporated into such a tool [14].

Software reuse can be practiced vertically or horizontally. Reusing components within a single domain –an area– is called as vertical reuse. On the other hand, horizontal reuse is the reuse of software components across different domains [5].

Opportunistic and systematic approaches are two basic forms of software reuse. In opportunistic reuse, new software is developed from existing systems, but they are modified to meet the requirements of the new software. In systematic reuse, new applications are developed from software that has been designed and developed to

reuse specially for other similar applications. Multi-project activities are generally carried out continuously over longer periods of time in an organization. Systematic reuse, the reuse of other software products, such as system designs and architectures, can increase the benefits of software reuse. This is called as “higher-level reuse” because system architecture is independent of code or language.

In the following sections, two software engineering methods related to these two approaches are introduced.

### **2.3. Component Based Software Engineering (CBSE)**

In the 1980s, object-oriented development methodologies that deal with the complexities of large system development became popular. These methodologies introduced “class” as basic unit of reuse. On the other hand, in the late 1990s, “software components” and “component based software engineering” were introduced in the concept of reusability.

Object-Oriented Techniques have been considered a powerful means of solving software crisis through their high reusability and maintainability. However, Object-Oriented Programming (OOP) has not brought many benefits, since they have not provided interoperability of components at the binary/runtime level. The paradigm for software development methods is shifting from OOP to Component-Based Development (CBD) and Component Based Software Engineering (CBSE) [15]. Software development through the planned integration of pre-existing software components is called CBD, CBSE, or simply componentware.

CBSE means developing systems by integrating pre-existing components. In development concepts, there are two basic corresponding approaches. In the top-down approach, customer requirements are taken into account. The system is decomposed continually into smaller parts until the level of detail is sufficient for an implementation with the help of existing components. On the other hand, in the bottom-up approach, existing reusable components are combined into higher-level components continually to meet the users’ requirements. Since requirements are not taken into account early in the bottom-up approach; the top-down approach is practical in most cases. However, if all the requirements are not known or they are inconsistent it becomes impractical [16].

The research on object-oriented technology and its intensive use by the industry have led to the development of component-oriented development. Rather than being an alternative to object-orientation, component technology extends the initial concept of objects. It stresses the desire for independent pieces of software that can be reused and combined in different ways to implement different software systems. It's clear that a strong movement toward component-oriented design and implementation. Some enterprises dictate, for example, that every Java class be designed and implemented as a Java Beans component which compromises to Java Bean specification. In these enterprises, programming groups exchange and share their software developments as Beans—they use each other's work in a plug-and-play application development setting.

While developing a system, firstly the requirements are analyzed. Secondly, system architecture is designed. Lastly, basic components are evolved. Components are combined to build the designed system architecture. In development process, after designing the system and identifying basic components, making a component or buying it is the main issue. If the available components meet the users' requirements, they are used. On the other hand, if they can be customized and adaptation cost of these components is less than building them from scratch, they are adapted.

Three major component architectures have been developed in the last years. Java Beans, ActiveX Controls and the CORBA Component Model intend to provide an infrastructure that will allow the construction of complex applications.

## **2.4. Domain Engineering (DE)**

The development context is expanded from a single project to multiple projects by reuse organization in a company. This organization is responsible for the coordination of the company's component repository.

A domain denotes a set of functional areas that exhibit similar functionality within systems. Within a domain, systems share many requirements. The acquired knowledge when building subsequent systems or components in the same domain is important for an organization. By reusing the assets developed by using the acquired domain knowledge in the development of new products, the organization will be able to deliver the new products in a shorter time and at a lower cost.

*Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable workproducts), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems [17].*

Domain Engineering is the part of software engineering that concentrates on general solutions for families of software systems with similar requirements and capabilities. While the traditional software engineering concentrates on satisfying the requirements for a *single* system, domain engineering concentrates on providing *reusable* solutions for *families* of systems. Domain Engineering addresses *multi-system scope* development [17].

Product families are scoped based on commonalities between the products in an organization. A software product line is a set of software systems that share common software architecture and a set of reusable components. In contrast to one system development at a time, strategic reuse is important. The *core asset developers* create these assets, such as the business case, requirements, software architecture. These assets change over time. This can cause problems in product production. [23] explains that changes can be managed in product lines. Software product line methods try to develop prefabricating software for multiple applications by providing configurable components. The ultimate goal of these methods is to create products efficiently. The *product developers* use the core assets to develop the products. When systems in the product line are known, software product line study begins by developing the core assets. This approach is called a *proactive* approach. Producing any product within the scope becomes a matter of assembling those assets. In some cases, software product line study begins with a few existing products. These products are used to generate the product line core assets. This approach is called a *reactive* approach. When the future is undefined, reactive product line approach is used to achieve flexibility [24].

Domain Engineering has three process components: *Domain Analysis*, *Domain Design* and *Domain Implementation*. The results of Domain Engineering are used in *Application Engineering* that develops software products from software assets.

### 2.4.1. Domain Analysis

*The systematic discovery and exploitation of commonality across related software systems are fundamental technical requirement for achieving successful software reuse. We define domain analysis as a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new system [18].*

Domain Analysis is a technic that meets this requirement. When the future requirements of multi-project applications are unknown, the risk of developing inappropriate software is high. This risk is reduced by Domain Analysis. An important difference of the domain analysis process from normal requirements engineering is the distinction of requirements into commonalities and variabilities. This is done in order to identify the common parts of a family of systems in a domain. These candidates are required for efficient reuse. Therefore, it is important to define the scope of the domain narrow enough in order to identify a large set of common requirements, which is the prerequisite of efficient domain engineering. Since it generalizes common features in similar application areas for identification of common objects and operations and describing their relations, experts in the software community generally agree that domain analysis is at the “heart of reuse” [5].

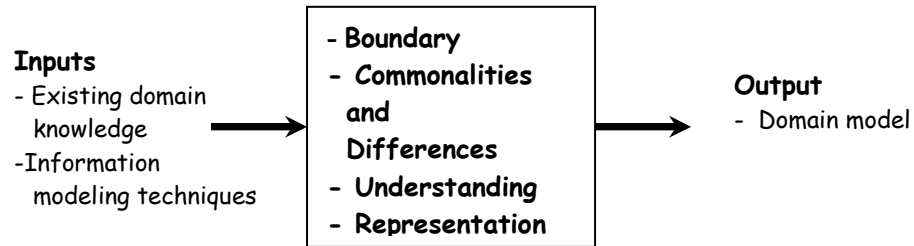
*Components that result from domain analysis are better suited for reusability because they capture the essential functionality required in that domain; thus, developers find them easier to include in new systems. We believe that domain analysis is a key factor in the success of reusability [19].*

Domain analysis activities consist of selection of a domain and analyzing concepts, properties, and solutions of the domain. Collected domain information and existing domain knowledge are used to get a model of the domain. The output of this process is domain the model, which can be reused. This domain model represents common and variable properties and dependencies between these properties in the domain.

These activities identify reuse opportunities and determine the common requirements of a family of systems in a domain. Identification, abstraction, and



encapsulation of objects in a particular domain are conducted [18]. The product of this phase is a domain model. Figure 2.1 represents inputs and output of domain design activities. Domain design activities look for a documented solution to the problem specified in a domain model.



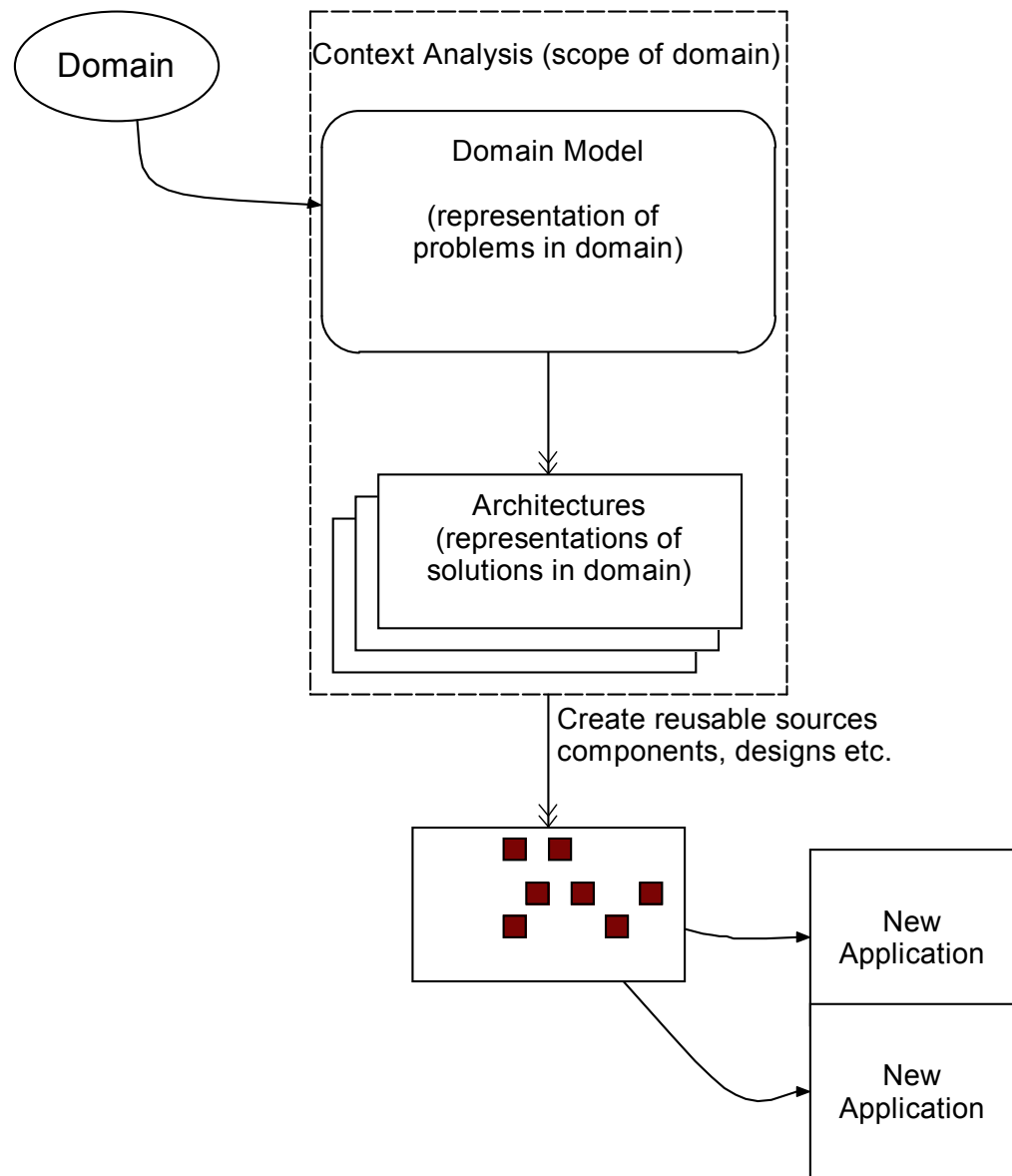
**Figure 2.1** Domain Analysis phase in Domain Engineering process

(Adapted from [27])

There are several research efforts in software engineering that are explicitly called domain analysis. A chronology is described in [18] and [20]. There are a large number of Domain Analysis and Domain Engineering methods. One of them, *Feature-Oriented Domain Analysis*, deserves special attention, since it is the most mature and best-documented method. In [21] and [22], this method is used with Domain Engineering process and analysis phase is documented. The method is briefly described in the following section.

Domain Analysis activity consists of three sub-activities to model the domain. These representations form a reference model for the systems in the domain: Context Analysis is an activity to be used to provide the boundaries of the domain. Domain Modeling represented in Figure 2.2 is an activity whose products describe the problem solved by the software in the domain. These products provide features of the domain, the domain dictionary and documentation of entities in the domain. Architecture Modeling is a phase that represents the structure of implementations in software. The

architecture model provides mappings with the domain model and guides the development of libraries of reusable components. Domain analysis methods provide specific representations to document the results of each of the domain analysis activities.



**Figure 2.2** Domain Analysis activity and sub-activities

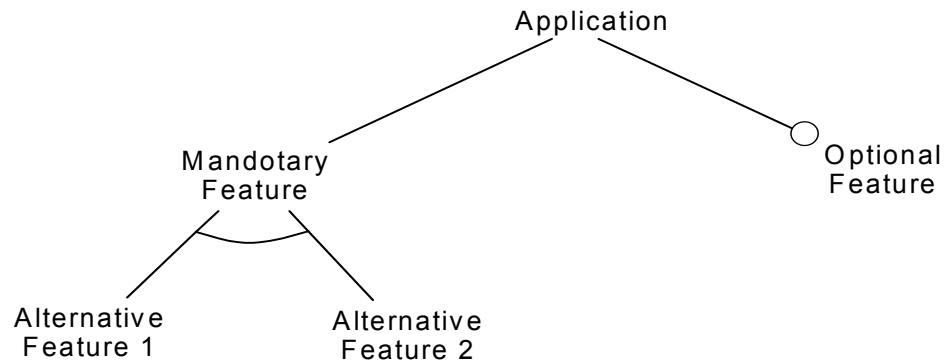
### ***Feature-Oriented Domain Analysis (FODA):***

The Software Engineering Institute at Carnegie Mellon University developed the Feature Oriented Domain Analysis (FODA) approach in 1990 [20]. It is based on identifying features of an application family. These features describe both system commonalities as well as differences within the domain and they relate to users. They present the characteristic of the domain. Feature models describe properties distinguishing between common and variable requirements. These features are, in essence, the requirements implemented for each of the systems in the domain.

Two basic phases that characterize the FODA process are FODA Context Analysis and FODA Domain Modeling. FODA Context Analysis defines the boundaries of a domain for analysis. The context model is used to determine that the application is within the domain of available domain products. FODA Domain Modeling provides a description of the problem space in the domain. Commonalities and differences of the systems in the domain are analyzed. Thus, a number of models representing different aspects of the problems are produced.

The three main tasks of domain modeling phase in FODA are feature modeling, information analysis, and operational analysis.

Feature modeling produces a hierarchical tree diagram in which features are typically one-word terms. Variability is modelled by permitting features to be mandatory, alternative and/or optional modeling elements those are represented in Figure 2.3. The feature model can be used by the requirements analyst to negotiate the capabilities of the application with the user, if the application is within the domain. Composition rules can also be defined between features.



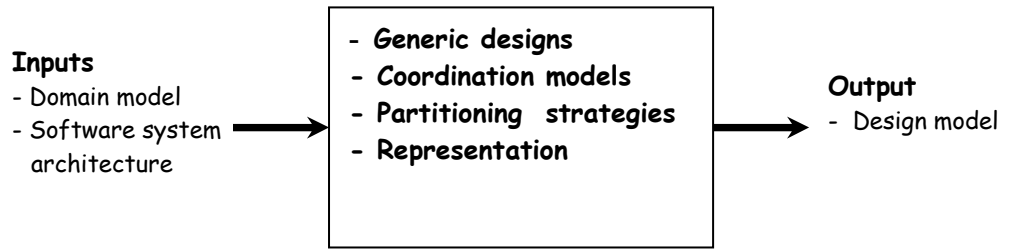
**Figure 2.3** Feature Model in FODA

Information analysis defines the data requirements of the applications family. The output of this task is a combination of entity-relationship diagrams, class diagrams, and structure charts. The information model can be used by a requirements analyst to acquire knowledge about the entities in the domain and their interrelationships.

Operational analysis identifies data flow of the applications of a family. The product of this stage includes data flow diagrams, class interaction diagrams and state transition diagrams. The operational model provides the requirements analyst with an understanding of the domain. The operational model provides the analyst with issues and decisions that cause functional differences between the applications.

### **2.4.2. Domain Design**

*Domain Design* activities look for a documented solution for the problem specified in a domain model. The process of developing a design model from the products of domain analysis and the knowledge gained from the study of software requirement/design reuse is called as Domain Design in Domain Engineering. The key point here is to capture a generic architecture that supports the reuse of components for the systems in the domain. This design model is a framework used in domain implementation phase to develop reusable components. In *Domain Design*, the domain model generated in domain analysis is used and architecture of the domain is generated. Therefore, the product of this phase is framework reusable software architecture. Figure 2.4 represents inputs, activities and outputs of this phase.

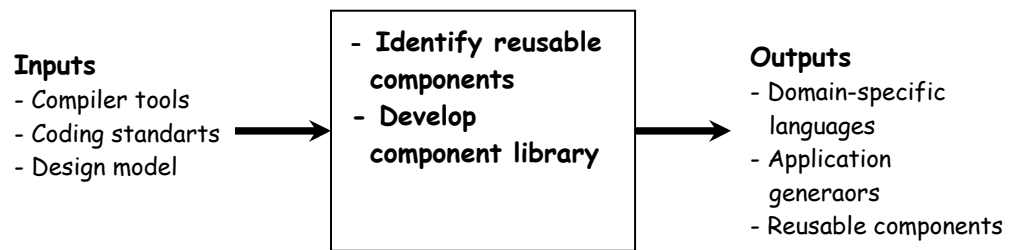


**Figure 2.4** Domain Design phase in Domain Engineering process

(Adapted from [27])

### 2.4.3. Domain Implementation

In *Domain Implementation*, components are implemented for the domain. Reusable components integrating the framework are constructed during the phase of domain implementation. This is the compositional approach of Domain Engineering. In the generative approach, Domain Engineering produces Domain Specific Languages (DSLs), which can be used as application generators to construct a family of applications in such a domain. Knowledge of the domain and design patterns are encoded in DSLs [25] [26]. In domain implementation, design model constructed in the domain design phase is used to develop components. Thus, the output of this phase is reusable components and DSLs. Figure 2.5 represents inputs, activities and outputs of this phase.



**Figure 2.5** Domain Implementation phase in Domain Engineering process

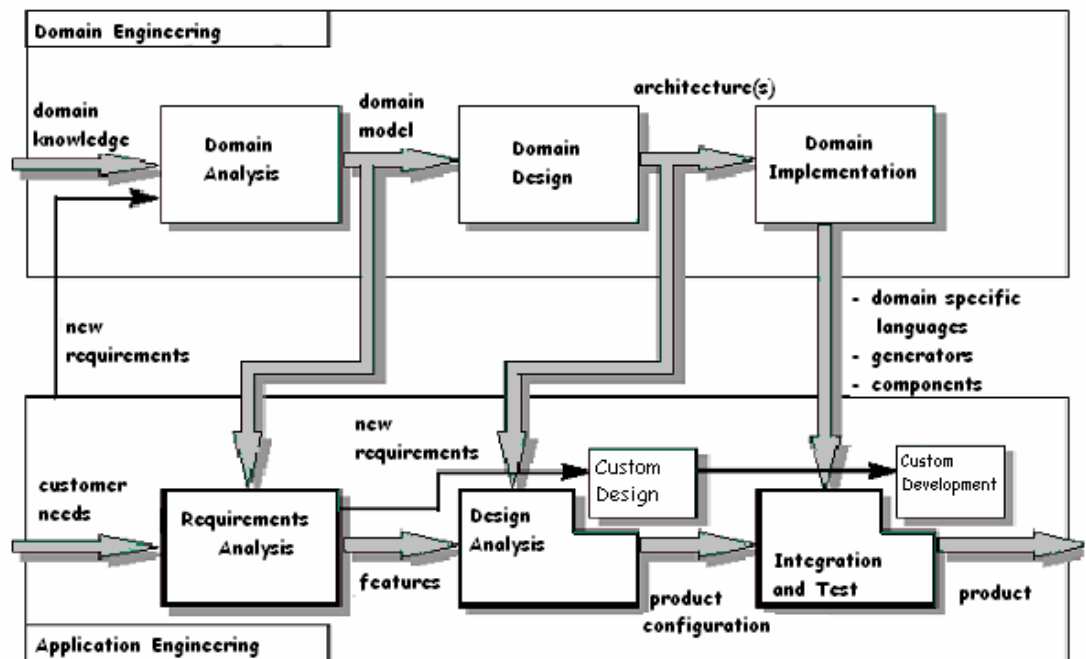
(Adapted from [27])

By making a reuse analysis in multiple projects of candidate components, the potential reuse degree of the component can be estimated. These components may be found by examining single in-house projects by domain analysis or by assessing the properties of components that are commercially available. In order to be usable in a multi-project context, in-house components are first tried for adaptation and generalization [16].

#### 2.4.4. Application Engineering

An *Application Engineering* process develops software products from software assets created by a domain engineering process. The results of Domain Engineering are reused during application engineering, i.e. the process of building a particular system in the domain. Figure 2.6 illustrates the relations between the domain engineering and the application development. Every new application is based on the architecture developed within the domain design activities. Domain engineering and application engineering are complementary interacting processes that comprise a model-based, reuse-oriented software production system [27]. In the application engineering process, during the requirements analysis for new systems, the customer requirements (*features*) are selected from the domain model. New customer needs not covered by a domain model are new requirements and those requirements require custom development. Therefore, Domain Engineering is a living process of creating and maintaining the reuse infrastructure. Finally, we assemble the application by using

the existing reusable components or using custom-developed components if they cost less according to the reusable architecture.



**Figure 2.6** Software Development Based on Domain Engineering

(Adapted from [17])

## 2.5. Java Archive (JAR) Library Files

Well-designed object-oriented software is decomposed into a multitude of classes. Normally in Java, each class is stored as a separate file having a .class extension. This is a problem when it comes to distributing programs. In addition to pure program files (the .class files), any non-trivial Java program also requires extra resource files: images, help files, configuration files, and so on.



The purpose of Java Archive (JAR) files is to pack more than one file into an archive. They are made easily by using the JAR Command-line tool. It is a Java-specific compression and decompression utility. Its main purpose is to take a list of files and compress them into a single JAR archive, or take a JAR archive and extract the compressed files from it. The JAR file format is compatible with the popular ZIP file format. The main difference between a plain ZIP file and a JAR file is that the JAR file format explicitly requires that the first file entry should be a file describing the contents of the JAR file. This file needs to have the following path: META-INF/MANIFEST.MF tree.

## 2.6. Java Beans Technology

Java Beans is *the software component* standard and architecture for the Java language platform [29]. They combine the benefits of Java (e.g., cross-platform development and execution) with the benefits of components (e.g., code reuse). A Java Bean is a reusable platform-neutral software component and reused everywhere. A Java Beans component conforms to a communication and configuration protocol, as prescribed by the Java Beans specification. The *Java Beans Specification* is the document that describes what a Java class must do to be considered a "Bean", how to use the classes and interfaces in the Java [28]. Three fundamental aspects of a Java Beans component as defined by the specification are events, properties, and methods.

To make a class into a component, the programmer must add functionality to the class. Java Beans turns classes into software components by providing several new features. Some of these features are specific to Beans. Others, like serialization, can apply to any class, Bean or otherwise. It is crucial to the understanding and use of Beans. Programmers can depend on any class that advertises itself as a Bean to conform to the rules set out in the specification. If it doesn't conform, the contract has been broken and the Bean is defective. Therefore, a Bean is a Java class that abides by a few relatively simple rules and design patterns:

- *Bean Creation*: Beans must have null constructors.
- *Event Handling and Bean Communication*: Beans communicate with each other by raising and listening to events, so they must define event interactions.
- *Persistence-Serilialization*: Beans must have persistence mechanisms.

- *Packaging*: Beans must be easy and efficient to distribute.
- *Introspection and BeanInfo*: Beans must provide mechanisms that enable a visual development tool to work out what methods and events the Beans have.

### Bean Creation

Beans can be resurrected from a persistent data stream. If we are sure that a bean is not of a saved version, it can be instantiated just like an object instantiation:

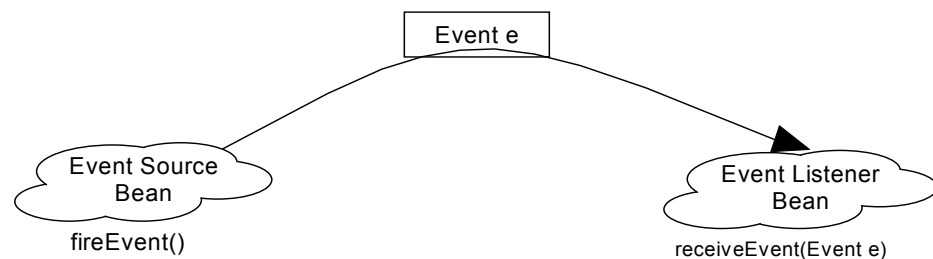
```
XBean aBean = new XBean ();
```

On the other hand, in a normal case, *instantiate* method is used. This method gives the system's infrastructure the chance to use a template-serialized Bean. Using this instantiate method creates a Bean using the default constructor. It restores its state data to the way it was when it was stored. This defines a null constructor:

```
XBean aBean = (XBean) Beans.instantiate (null,"XBean");
```

### Event Handling and Bean Communication

Beans communicate with other software components by a way of firing and receiving events. This mechanism can be thought as the pins of an IC in the electronics world. Some pins are for output just like event firing, while others are for input just like event receiving. The mechanism is represented in Figure 2.7.



**Figure 2.7** Event Handling in Java

In a Java Beans based system, event-based communication is an active process for the event source and a passive process for the listener. A bean should fire events when it wishes to transfer anything to the outside world. On the event receiving side, if a bean wishes to receive information from the outside world it should be a listener for some event type. Beans are source objects in this mechanism. Typically, a Bean does its work and, then sends notifications to all registered targets. Interested parties register with the source object for event notifications and perform their own operations in response to these events.

An event can only be heard by a listener if:

- Listener implements the *EventListener* interface
- Listener tells the Event source that it is interested in hearing about the event by *registering* itself using an *addEventListener ()* call

In many cases, a Java Beans component will function as a source for certain types of events, yet be capable of registering as a target for events produced by other components.

### **Persistence – Serialization**

A persistence mechanism allows a Bean or a collection of Beans to be saved in a file. To provide persistence, beans use Java serialization mechanisms. In Java serialization, all the member variables of an object can be made persistent by the Java run-time. Bean creator implements the *java.io.Serializable* interface to provide persistence.

### **Packaging Beans**

A single Java bean is usually composed of at least three files. Those files correspond to:

- A Bean class.
- An associated BeanInfo class.
- An iconic representation for the bean; that is, a GIF file.

Because beans are supposed to be complete and inseparable entities, having beans composed of several files can cause a lot of problems if a file gets lost or corrupted or simply stored in the wrong place. Therefore, a Bean is packaged by putting it into a JAR file. The JAR can contain not only the Bean's own class file but also other classes that the Bean uses, icons, graphics, internationalized text, and HTML-format help files.

JAR files holding Java Beans use beans-specific pairs. When a JAR file is used to distribute beans, each bean needs an additional attribute-value pair in the MANIFEST.MF file:

Java-Bean: True

If this additional line is not present, the bean will not be visible to the tool or application that processes the JAR file. JAR tool knows nothing about Java Beans; hence it does not add this line by default.

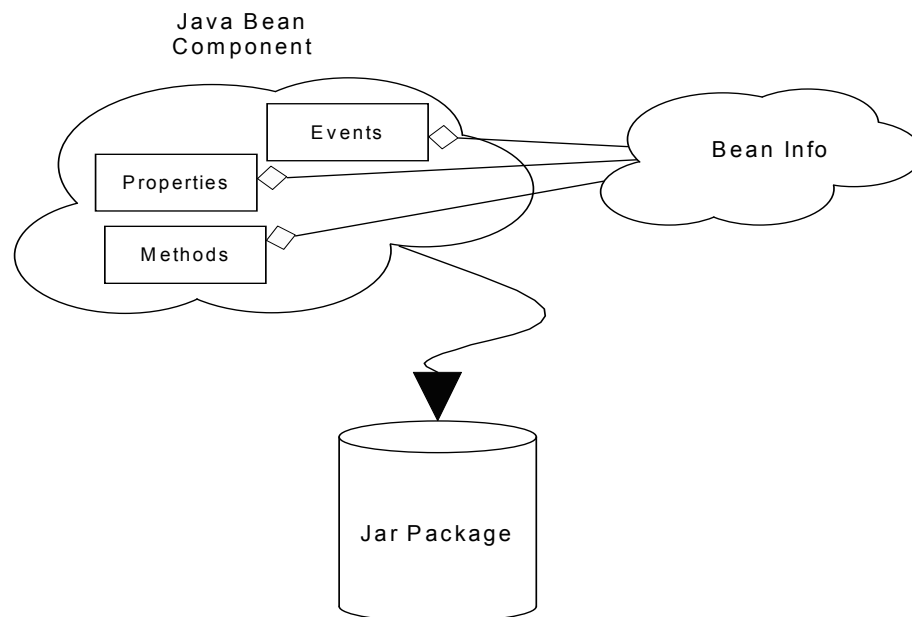
### **Introspection and BeanInfo**

Process of discovering an object's characteristics is called *introspection*. The Java Beans framework provides a class that is called Introspector. An Introspector object assists in discovering a Bean's configurable characteristics. It provides functionality to discover information about a Bean from a complementary Bean configuration class that optionally accompanies each Bean. This complementary, support class is called the *bean-info* class. The Java Beans framework provides the interface BeanInfo, which describes the services that a bean-info class implements. An Introspector object manipulates and makes available a Bean's configuration services in a general-purpose manner using the BeanInfo interface. A Bean publishes its configuration support via methods in its bean-info class. A Bean analyzer then instantiates the bean-info class and queries the appropriate method during the Bean configuration process.

The Java Beans standard is a low-level software component standard tailored to Java. As such, it concentrates on how Java software components, informally called beans, should present their black box interface to the outside world, and more specifically toward tools and designers who rely on beans for their applications. All Java Beans present a black box interface that consists of properties, event-firing outputs, and plain public methods that can be called on a bean. Because beans are

meant to be software components, their design and implementation should begin with the fundamental assumption that a bean can be deployed in any application environment.

Bean-Info classes explicitly control the set of properties, events, and methods that a bean exports. If a class XBeanInfo accompanies any bean class X, then bean environments recognize this as a formal relationship. Bean X is said to have an associated BeanInfo class. BeanInfo class associates either 16 by 16 or 32 by 32 pixel color or black-and-white icons with a bean. All this can be achieved by implementing the BeanInfo interface, in a class ending with *-BeanInfo*. Figure 2.8 illustrates the relation between Java Bean component, Bean Info and JAR files:



**Figure 2.8** Java Beans and JAR Packages

Using lists of properties is not always the best way to handle customizing Beans. Some Beans are too complex to be easily manipulated in this way. Sometimes the developer who uses the Java Beans architecture simply needs total freedom to

design a property editor for one or more, possibly specialized, properties. In this case, the Java Beans framework allows the developer to design and register a custom, graphical object, as a collection of Graphical User Interface (GUI) components in a container (panel). A Beans developer can embed a property sheet into the Bean itself, and the Integrated Development Environment (IDE) then uses this "customizer" to customize the Bean.

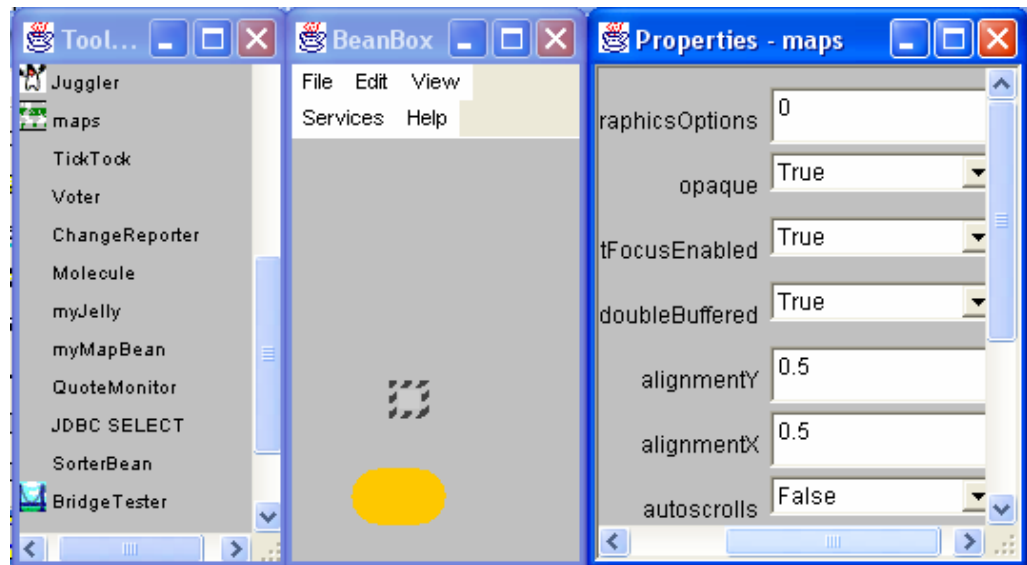
There is some additional overhead in designing classes as Beans. For example, the fundamental Bean requirement of a no-argument constructor implies that some programmers must change their design habits, or in some environments, no-argument methods must be provided to connect Beans together graphically, whereas in traditional source code-level method invocation has no limits.

Java beans are also tool-friendly in that they are only used at design-time; aspects such as a bean palette icon, an explicit description of the bean's interface. The tool, Bean Development Kit, can test Java Beans. In the next section, contents of the BDK tool are presented.

## **2.7. Bean Development Kit (BDK) and BeanBox Testing Application**

Bean Development Kit (BDK) is a reference environment to develop and test beans. JavaSoft produces the BDK. Bean-Box is the environment that we can drop our beans and exercise them in various ways. The smaller ToolBox window represents a bean palette from which we can pick any bean to work with, as if from a physical box full of components. The ToolBox window is where the BeanBox application lists all the beans that you may select for use within Java Beans-based application. The last window called PropertySheet lists the currently selected bean's properties and their values. These environments are illustrated in Figure 2.9.

To test the beans, JAR file of the bean is copied to the JARs directory under BDK directory. When the tool is run, beans are loaded and shown on the ToolBox palette. Beans are dropped on to the BeanBox canvas. More beans can be put on this canvas and linked together. Source beans and listener beans are connected and their behaviours are tested.

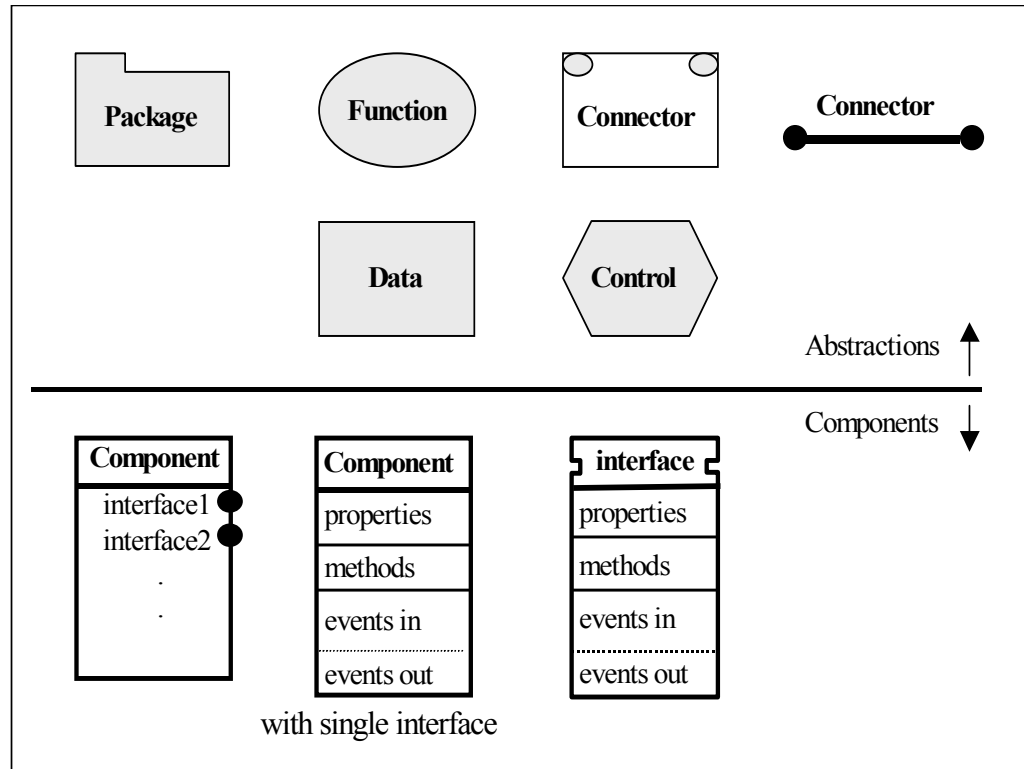


**Figure 2.9** Bean Development Kit (BDK)

BDK also includes some bean archive files in the JAR directory. Source codes of these beans can be examined.

## **2.8. COSE Modeling Language (COSEML) and COSECASE**

For the component oriented software engineering, COSE Modeling Language (COSEML) has been designed and a tool (COSECASE) has been developed. A detailed definition of the modeling language and the tool can be found in [8]. Graphical representations of the modeling elements are shown in Figure 2.10.



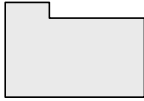



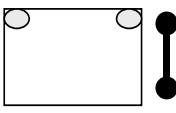

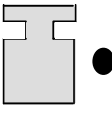



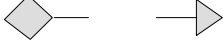
**Figure 2.10** Graphical symbols in COSEML

System is designed in two levels in COSEML: abstraction level and implementation level. Therefore, the model can simultaneously contain abstraction level and implementation level components. Abstract level includes Package, Data, Function, and Control abstractions. “Packages” are represented by Unified Modeling Language’s (UML) package symbol. Data, Function, and Control abstractions can be included in a Package. Implementation level constructs are mainly “components”. A Component corresponds to the existing implemented component code. Each component has interfaces to represent its properties, methods, and events. A component represents its services by its interfaces. Therefore, an Interface is the connection point of a component and services requested from a component are invoked through this interface.

Table 1 presents all graphical symbols in COSEML with further details.



**Table 1** COSEML symbols and their meanings.

Symbol	Explanation
	<b>Package:</b> Package is for organizing part-whole relations. A container that wraps system-level entities and functions etc. at a decomposition node. Can contain further package, data, function, and control elements. Also can own one port(s) of one or more connectors. Can be implemented by a component. The contained elements are within the scope of a package; they do not need connectors for intra-package communication.
	<b>Function:</b> Function abstractions represent a system-level function. Can contain further function, data, and package elements. Can own connector ports. Can be implemented by a component.
	<b>Data:</b> Data abstractions represent a system-level entity. Can contain further data, function, and package elements. Can own connector ports. Has its internal operations. Can be implemented by a component.
	<b>Control:</b> Control abstractions correspond to a state machine within a package. Meant for managing the event traffic at the package boundary, to affect state transitions, as well as triggering other events. Can be represented by a component.
	<b>Connectors:</b> Connectors represent data and control flows across the system modules. Cannot be contained in one module because two ports will be used by different modules. Ports correspond to interfaces at components level.
	<b>Component:</b> A Component corresponds to the existing implemented component codes. Contains one or more interfaces. Can contain other components. Can represent package, data, function, or control abstraction.
	<b>Interface:</b> An Interface is the connection point of a Component. Services requested from a component have to be invoked through this interface. A port on a connector plugs into an interface.
	<b>Represents Relation:</b> A Represents relation indicates that an abstraction will be implemented by a component.
	<b>Event Link:</b> An Event link is connected between the output event of one interface and the input event of another. The destination end can have arrows corresponding to the synchronization type.
	<b>Method Link:</b> A Method link is connected between two interfaces to represent a method call. Arrow indicates message direction.
	<b>Composition and Inheritance Relation:</b> UML class diagram relations are utilized. Diamond: Composition, Triangle: Inheritance.

In [7] a process model, COSE Process Model is presented for component-oriented development. This model represents a complete, consistent, and clear guide to software developers to develop component oriented software systems. COSE Process Model building activity uses top-down approach to iteratively decompose the system into nearly independent partitions as mentioned in [16]. These partitions are represented with corresponding software components. System development is done by integration rather than coding from scratch. This provides both concurrency and manageability in system development. Since the problem is divided into relatively small sub problems, the model provides both concurrency and manageability in system development.

# **CHAPTER 3**

## **APPLICATION OF DOMAIN ENGINEERING PROCESS TO GEOGRAPHICAL INFORMATION SYSTEMS (GIS) DOMAIN AND DOMAIN FRAMEWORK**

Geographical Information Systems (GIS) in general correspond to geographical data. Much of current progress in GIS is towards making it easier for users to construct maps. Geographic data is thought as layers of information underneath the computer screen [30]. Therefore, a GIS can present many layers of different information. One layer could be made up of all the roads in an area and another one could represent all the lakes in the same area. GIS is used for every specialization of the defense industry in many nations around the world. All projects in a military company have a GIS part to show activity in the tactical area. A lot of effort is spent to develop GIS applications in such projects. In order not to develop GIS applications from scratch, a framework is intended to be developed, which utilizes reusable components.

In this part of the thesis, a Domain Engineering process has been applied to GIS domain, which is a general-purpose environment, for use in an organization:

1. Requirements were collected for the GIS domain and existing domain knowledge was collected from the experts.
2. Domain was simply analyzed and a feature model of the domain was developed for the *Domain Analysis* phase of the process.
3. Architecture model of the domain was constructed for the *Domain Design* phase of the process.

4. Components were implemented into the *Domain Implementation* phase of the process. Reusable components integrating the framework were constructed during this phase. These components were found by examining single in-house projects by domain analysis, or by coding, or by assessing the properties of components commercially available in the market. These components were made as Java Bean components and packaged into Java Archive (JAR) files for deployment. In the following sections, these processes are mentioned in a detailed manner.

Components developed are used for *Application Engineering* process that develops software products from software assets created by a domain engineering process. These components are imported into COSECASE to enhance the tool for GIS application designs. To import these components into COSECASE, an interface layer was developed. Next chapter is adapted for implementation of this issue.

### **3.1. Domain Analysis for GIS Domain**

GIS Domain framework offers some services GIS part of the product line of the company needs. These services meet all the requirements of the applications in the domain. These functions are:

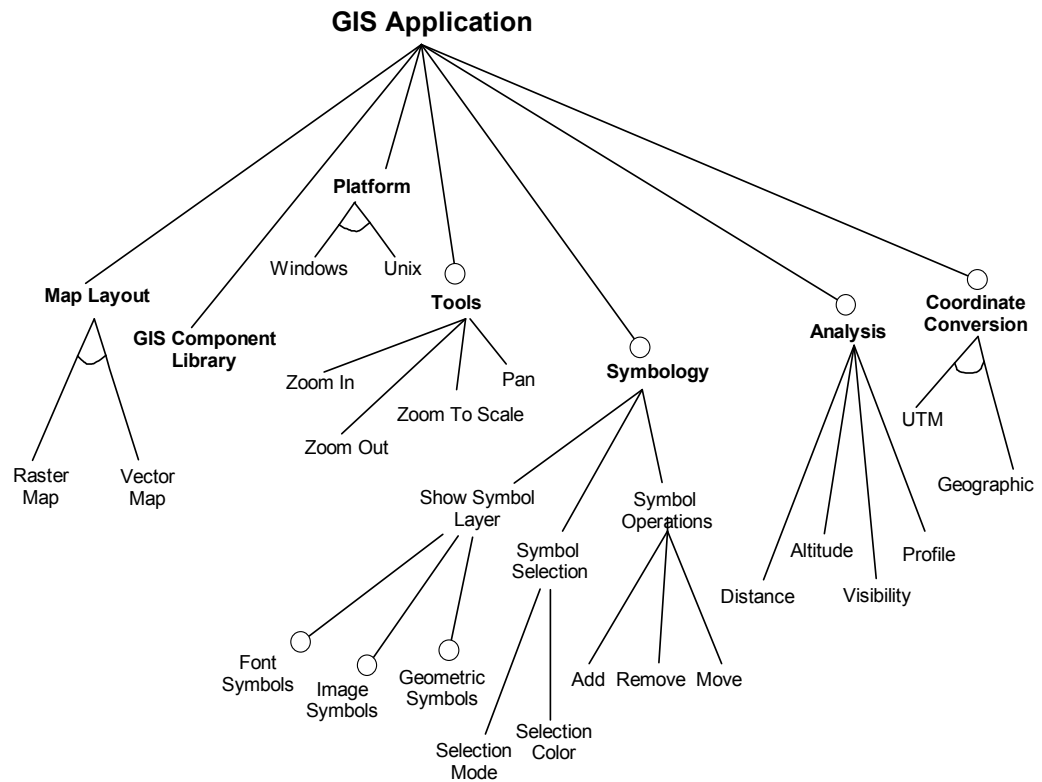
1. Applications will be developed on both Unix and Windows platforms. Therefore, components will be developed by a language, which runs on both platforms. This language is selected as Java. JavaBeans will be used to develop components.
2. GIS Applications will use multi-layered structure. More than one layer will be shown on the screen at a time.
  - Layers can be added to the map and removed from the map dynamically.
  - Visibility of the layers can be changed at runtime.
  - Raster data can be shown on a layer.
  - Vector data can be shown on a layer.
  - Symbols specific to the application can be shown on a layer.

3. Custom GIS operations can be achieved easily. These operations include:
  - Zoom In / Out operations: These operations are used to zoom in or zoom out layers. Operations are processed according to some criteria:
    - By using scale
    - By selecting a rectangular area
    - By entering ratio
  - Pan Operations: These operations are used to pan the layers. Operations are processed according to some criteria:
    - By selecting a point on the layers
    - By specifying direction and distance
4. Analysis operations can be conducted on the layers by using 3D data formats. 3D formatted files will be in Digital Elevation Model (DEM) format. These operations are:
  - Distance: This operation is used to measure the distance between points.
  - Altitude: This operation is used to measure the altitude of a point.
  - Visibility: This operation is used to analyse visibility of a point. Painting the visible points on a layer will show result of the analysis.
  - Profile: This operation is used to process profile analysis of selected points. The result of the operation will be shown on a window. This window will be embedded in the GIS component library.
5. Military Symbology can be displayed on the layers.

- Symbols specific to the application can be displayed on a layer. These symbols can be in the form of:
    - Geometrical shapes: These shapes are point, line, rectangle, ellipse, polygon, and polyline.
    - Fonts: Fonts can be displayed in different sizes.
    - Pictures.
  - Symbols shown on the layer can be selected. Selected shapes will be shown in another color. Drawing color and selection color can be set by the application.
  - Symbols can be moved around in a layer.
6. Conversion between coordinate systems: Basically, two types of coordinate systems will be used. Conversion between these coordinate systems will be conducted. In the future, other conversion mechanisms related to different coordinate systems will be added to the library. Currently supported coordinate systems are:
    - Universal Transversal Merkator (UTM)
    - Geographic
  7. Getting real world coordinates from the map: Component library will read real world coordinates from the map.
  8. Getting and setting scale of the map: Scale of the map can be set and get by the applications.
  9. Getting and setting extent of the map: Extent of the map can be set and get by the applications.
  10. A component library commercially available in the market will be used to handle basic GIS operations. This library can be changed according to performance problems. In the future dependency of this component library will be minimized.

11. Different raster maps will be loaded automatically according to map scale. For example, when the current map scale is between 1/350.000 and 1/700.000, raster maps with 1/500.000 scale will be loaded. Basically eight-type raster map scale is used and these maps are loaded automatically.

After these requirements and domain knowledge were collected from domain experts and the product line, domain was modelled by using Feature Oriented Domain Analysis (FODA) feature model. This model is shown in Figure 3.1.



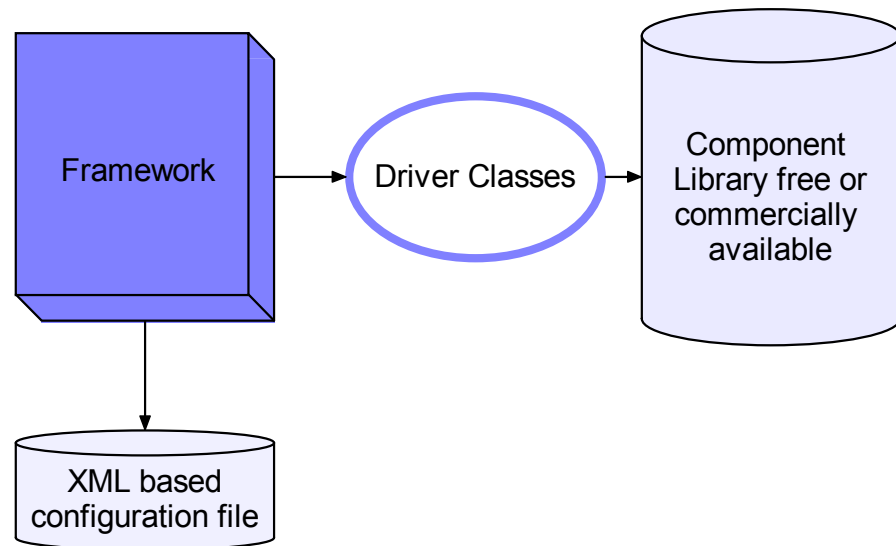
**Figure 3.1** A Feature Model for the GIS Domain

Some composition rules were defined on the feature model:

- “Tools” requires “Map Layout”
- “Analysis” requires “Map Layout”
- “Coordinate Conversion” requires “Raster Map Layout”

### 3.2. Domain Design for GIS Framework

In *Domain Design*, the domain model generated in domain analysis is used and architecture for the domain is generated. This GIS framework covers the requirements of the GIS application domain in the company. It defines most of the component structures that are used in GIS applications to meet application requirements. One of the requirements obtained from the domain analysis phase is making use of a GIS component library that is commercially available in the market. This library should be changed according to any specific application without changing and compiling the components in the framework. To solve this problem, a class called “*driver*” and an eXtended Markup Language (XML) - based file are used. Figure 3.2 illustrates the mechanism.



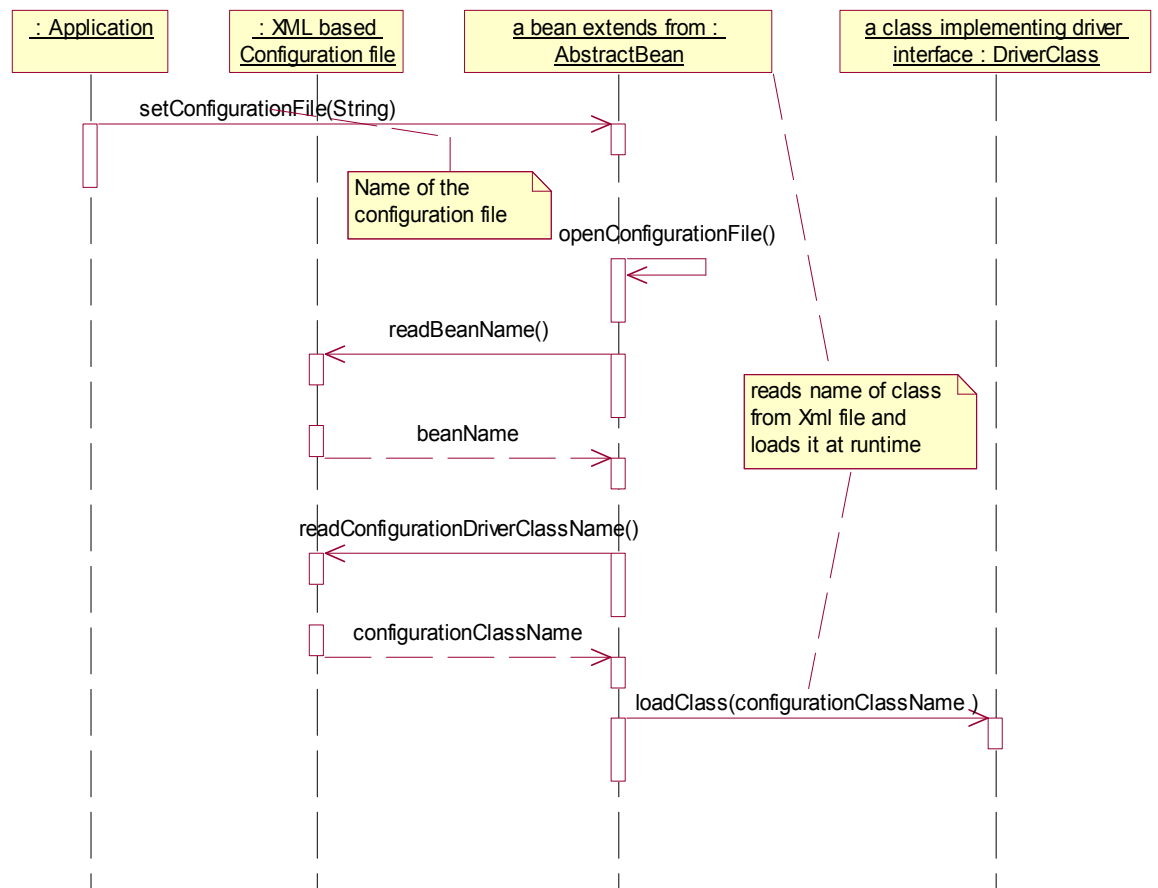
**Figure 3.2** Using a component library within the GIS Domain Framework



This driver class implements required methods by using methods of the component library. GIS Domain framework does not initially know about the driver class. The GIS framework represents interfaces to be implemented for the driver classes. Developer of the driver class implements these driver interfaces by methods of the component library and inserts the name of the driver class to the XML based configuration file. Structure of the configuration file is shown below for a coordinate converter component:

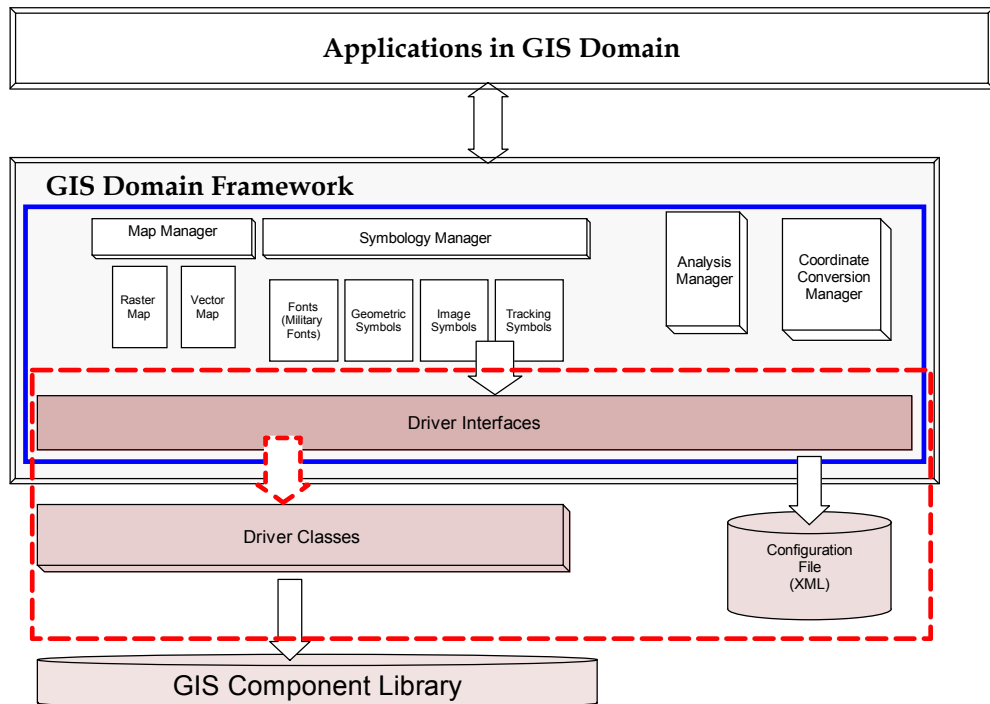
```
<ConfigurationFile>
  <BeanCallDefinitions>
    <BeanName>CoordinateConversion</BeanName>
    <BeanDriverClass> GISFramework.MOJDriverCoordinateConversion</BeanDriverClass>
  </BeanCallDefinitions>
</ConfigurationFile>
```

The name of the driver class implementing related driver interface is read from the configuration file by components in the framework. Driver class is loaded at runtime by using Java Reflection package. Components always call methods of the driver interfaces represented in the framework after loading the driver class. The process of loading a driver class by the framework at runtime is shown in Figure 3.3. Setting the new configuration file changes the component library at run time.



**Figure 3.3** Setting configuration file in GIS framework

Domain architecture was constructed in this phase. GIS Framework was designed considering the issues mentioned above. Interfaces for beans, beans and interfaces for driver classes were designed. Figure 3.4 shows this architecture and detailed descriptions about the elements of the architecture are given below. In additiond, class diagrams were modeled with UML and sequence charts of components were modeled in Rational Rose tool. This documentation is placed in Appendix B.



**Figure 3.4** GIS Domain Framework design

Projects in the company are developed on both Unix and Windows platforms. To meet multiplatform requirements, the component library was developed by using Pure Java.

Components of this architecture are mentioned below in a detailed manner:

*Commercially Available Components:* In the projects that include GIS applications, all requirements were met by using appropriate component library from the market and in-house codes formerly developed by software engineers in the company.

*Driver Interfaces:* Company does not want to depend on a specific component library from the market. Since, performance of the component library may not be good enough in some circumstances. In such a case, company wants to achieve the capability to change the component library or to use its own components doing the same work. Therefore, storing the component library, e.g. Map Objects Java, to codes in the framework, and compiling the framework is not a good solution. Flexibility in using component library is another requirement of the GIS domain framework. To find a resolution for this requirement, the framework offers some interfaces called “driver interfaces”. With such an approach, components in the framework always call the functions in these driver interfaces defined statically.

*Driver Classes:* Classes will implement “driver interfaces” by using the functions of the selected component library, for example Map Objects Java. The classes implementing the driver interfaces are developed by using the methods of the selected component library once for each component library. Hence, a project can use any component library, or projects on the domain can use a different component library at any time. In addition, changing component library does not require changing and compiling the framework. Therefore, when the component library is changed, there is no need to recompile the framework. Driver classes are loaded at runtime. The name of the driver class implementing the selected component library is given to the framework by the application. Framework reads the configuration file and loads the class at runtime. An application can change the component library at runtime.

*XML Based Configuration File:* This file contains some application specific properties. Firstly, this file includes the name of the driver class. Then, it includes component specific properties, such as paths of the maps.

*Map Manager:* This is an abstraction for the designers. Its task is to show raster and vector maps. Raster and Vector beans are handled in this part of the architecture.

*Symbology Manager:* This is an abstraction for the designers. Its task is to show layers on which symbols are shown. Layers showing figures that include fonts, pictures, geometric shapes and tracking symbols are managed in this part of the architecture.

*Analysis Manager:* Its task is to make analysis operations and show analysis results. This manager refers to the analysis bean.

*Coordinate Conversion Manager:* Executes coordinate conversion operations. This manager refers to coordinate conversion bean.

### **3.3. Components for GIS Framework Domain Implementation**

In *Domain Implementation*, components were implemented for the domain. Components have been developed as Java Beans in this framework. Therefore, this framework consists of Java Beans as components to meet requirements in the domain and is based on interfaces, drivers, beans and communication of beans by listener and event mechanism in Java. These components have been identified by examining in-house projects by domain analysis, or by coding, or by assessing the properties of components in the component market. These components have been developed as Java Beans and packaged into JAR libraries for deployment and have been tested by Bean Development Kit (BDK) tool.

A naming convention is used in developing elements in the framework for code generation puposes. All elements of the framework conform to this naming convention. It simplifies the understanding of the element types, such as abstract classes, listeners, and beans from the name. In addition, events of the beans and code generation mentioned in the next section are conducted simply by this mechanism. This naming convention is as below. The first three are naming convention rules for the base classes and others are rules and names of the beans included in the framework.

1. All abstract classes are named “AbstractBean + Name” in the framework.

- a. “AbstractBean” class is an abstract class for all bean classes. They are inherited from this class. It contains implementation of some common functions that a bean must have.
  - b. “AbstractBeanLayer” class is abstract class for all layer beans. They are inherited from this class. It contains implementation of some common functions that a layer bean must have.
  - c. “AbstractBeanLayerSymbols” class is abstract class for all symbol layer beans. It contains implementation of some common functions that a symbol layer bean must have.
2. All interfaces for the abstract classes are named “InterfaceBean + Name” in the framework.
  - a. “InterfaceBean” is the interface for AbstractBean class in the framework.
  - b. “InterfaceBeanLayer” is the interface for AbstractBeanLayer class in the framework.
  - c. “InterfaceBeanLayerSymbols” is the interface for AbstractBeanLayerSymbols class in the framework.
3. All exception classes for abstract classes are named as “ExceptionBean + Name” in the framework.
  - a. “ExceptionBean” is the exception for AbstractBean abstract class.
  - b. “ExceptionBeanLayer” is the exception for AbstractBeanLayer abstract class.
  - c. “ExceptionBeanLayerSymbols” is the exception for AbstractBeanLayerSymbols abstract class.
4. All non-abstract bean classes are named “Bean + Name” in the framework. The names for non-abstract bean are as follows:
  - a. BeanMap
  - b. BeanTool

- c. BeanLayerVector
- d. BeanLayerRaster
- e. BeanLayerAnalysis
- f. BeanLayerSymbolsFont
- g. BeanLayerSymbolsGeometry
- h. BeanLayerSymbolsDrawing
- i. BeanLayerSymbolsImage
- j. BeanTrackingFontLayer
- k. BeanTrackingGeometriesLayer
- l. BeanTrackingImagesLayer
- m. BeanCoordinateConverter

5. All interface classes are named “Interface + Name” in the framework. The names for interfaces of beans are as follows:

- a. InterfaceMap
- b. InterfaceTool
- c. InterfaceLayerVector
- d. InterfaceLayerRaster
- e. InterfaceLayerAnalysis
- f. InterfaceLayerSymbolsFont
- g. InterfaceLayerSymbolsGeometry
- h. InterfaceLayerSymbolsDrawing
- i. InterfaceLayerSymbolsImage
- j. InterfaceCoordinateConverter

6. All exception classes for beans are named “Exception + Name”. The names for exceptions classes are as follows:

- a. ExceptionMap
- b. ExceptionTool
- c. ExceptionLayerVector
- d. ExceptionLayerRaster
- e. ExceptionLayerAnalysis
- f. ExceptionLayerSymbolsFont
- g. ExceptionLayerSymbolsGeometry
- h. ExceptionLayerSymbolsDrawing
- i. ExceptionLayerSymbolsImage
- j. ExceptionCoordinateConverter

7. All listener classes are named as “Listener + Name”. In addition, the methods of subscribing and unsubscribing listeners are named as “add + ListenerName (ListenerName) + Listener” and “remove + ListenerName (ListenerName) + Listener”. This naming convention is convenient to the naming conventions of Java Beans.

- a. ListenerMap: The class that implements this interface listens to the Map bean.
  - i. Subscribing listener to the map bean:  
addListenerMapListener(ListenerMap x)
  - ii. Unsubscribing listener from the map bean:  
removeListenerMapListener (ListenerMap x)
- b. ListenerTool: The class that implements this interface listens to the Tool bean.



- i. Subscribing listener to the tool bean:  
addListenerToolListener(ListenerTool x)
    - ii. Unsubscribing listener from the tool bean:  
removeListenerToolListener (ListenerTool x)
  - c. ListenerLayerVector: The class that implements this interface listens to the Vector Layer bean.
    - i. Subscribing listener to the vector layer bean:  
addListenerLayerVectorListener (ListenerLayerVector x)
    - ii. Unsubscribing listener from the vector layer bean:  
removeListenerLayerVectorListener (ListenerLayerVector x)
  - d. ListenerLayerRaster: The class that implements this interface class to the Raster Layer bean.
    - i. Subscribing listener to the raster layer bean:  
addListenerLayerRasterListener (ListenerLayerRaster x)
    - ii. Unsubscribing listener from the raster layer bean:  
removeListenerLayerRasterListener (ListenerLayerRaster x)
  - e. ListenerLayerAnalysis: The class that implements this interface listens to the Analysis Layer bean.
    - i. Subscribing listener to the analysis layer bean:  
addListenerLayerAnalysisListener (ListenerLayerAnalysis x)
    - ii. Unsubscribing listener from the analysis layer bean:  
removeListenerLayerAnalysisListener(ListenerLayerAnalysis x)
8. All driver interfaces that are deployed are named Driver + Name. The names for driver interfaces are as follows:
- a. Driver
  - b. DriverMap

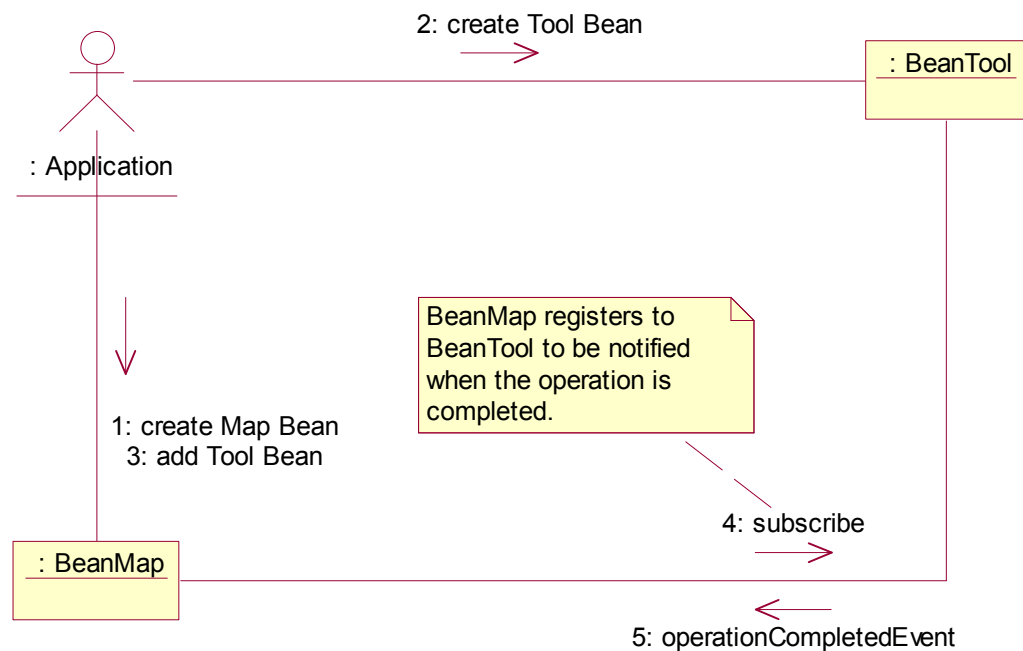
- c. DriverTool
- d. DriverLayerVector
- e. DriverLayerRaster
- f. DriverLayerAnalysis
- g. DriverLayerSymbolsFont
- h. DriverLayerSymbolsGeometry
- i. DriverLayerSymbolsDrawing
- j. DriverLayerSymbolsImage
- k. DriverCoordinateConverter

Beans communicate with each other by using listeners and events. Outside world uses services in these beans by invoking methods of these beans. Beans register themselves with other beans to be informed when an action happens or some action is requested from the source bean by using listener interfaces. Java Bean Components in the framework are explained below:

- a. *BeanMap*: Describes functionalities of the “map” component in this framework. It implements “InterfaceMap” interface that must be implemented by beans written as the map component in this framework.

A Map Bean is the core bean in this framework. All other beans are added to the Map Bean. Beans that want to listen to Map Bean implement ListenerMap interface to be notified with the events occurring in the bean. They register themselves with the Map Bean automatically, when they are added to the Map Bean. Map Bean notifies its listeners when the scale of the map changes and coordinate unit of the map is set. A Map Bean listens to the Tool, Raster Layer, Vector Layer, Analysis Layer, and Coordinate Converter beans. Therefore, the Map Bean implements listener interfaces of these beans, which are “ListenerTool”, “ListenerLayerRaster”, “ListenerLayerVector”, “ListenerLayerAnalysis”, and “ListenerCoordinateConverter”. When these beans are added to the Map Bean, it registers itself with these beans automatically.

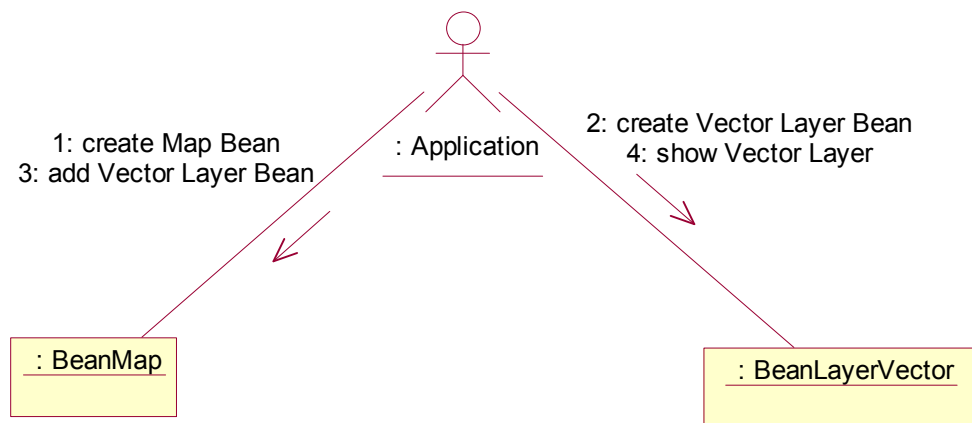
b. *BeanTool*: Describes functionalities of the “tool” component in this framework. It implements “InterfaceTool” interface that must be implemented by the Tool Bean in this framework. Beans that want to listen to Tool Bean implement “ListenerTool” interface to be notified with the events occurred in this bean. Tool Bean notifies its listeners when zoom, pan, distance and unsetting requests are taken place. Figure 3.5 illustrates the registration mechanism of the Tool Bean with the Map Bean that is the core bean in this framework.



**Figure 3.5** Communication between the BeanMap and the BeanTool components in the GIS Domain Framework

c. *BeanLayerVector*: Describes functionalities of the “vector layer” component in this framework. It implements “InterfaceLayerVector” interface that must

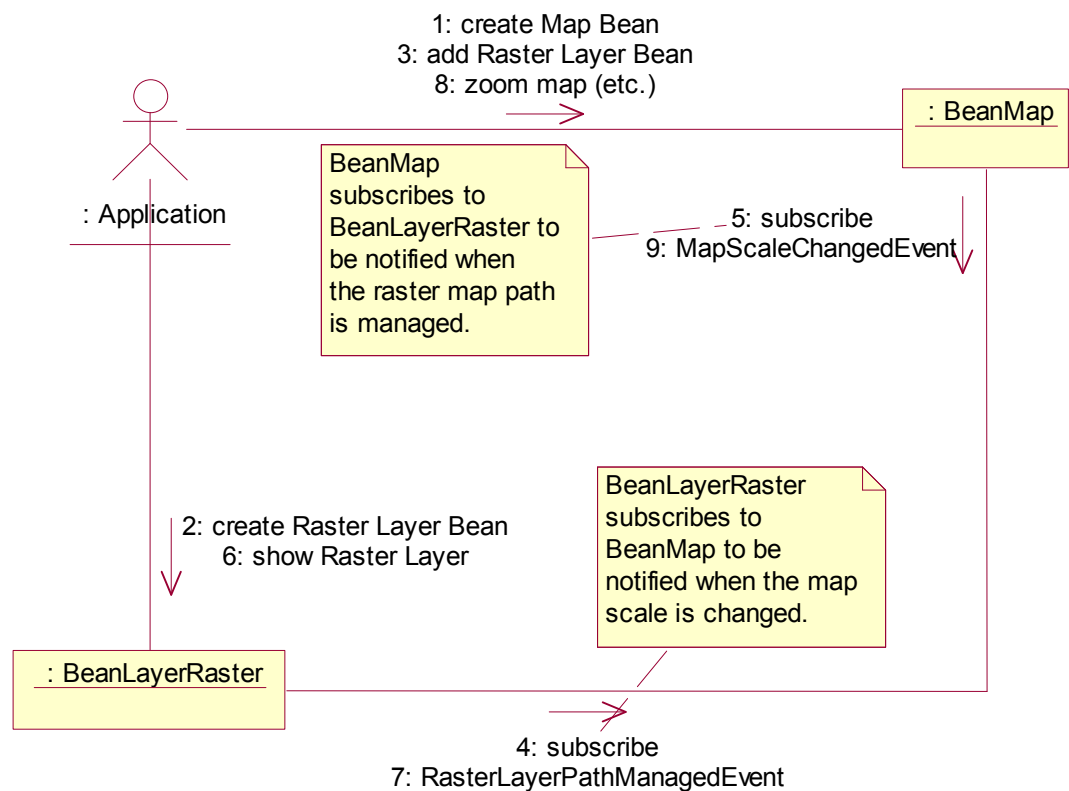
be implemented by beans written as Vector Layer Bean in this framework. Beans that want to listen to the Vector Layer Bean implement “ListenerLayerVector” interface to be notified with the events occurred in the bean. When Vector Layer Bean is added to map bean, it registers itself with the Map Bean automatically. The Vector Layer Bean notifies its listeners when visibility of the vector map is set from outside. Figure 3.6 illustrates the registration mechanism of the Vector Layer Bean with the Map Bean that is the core bean in this framework.



**Figure 3.6** Communication between the BeanMap and the BeanLayerVector components in the GIS Domain Framework

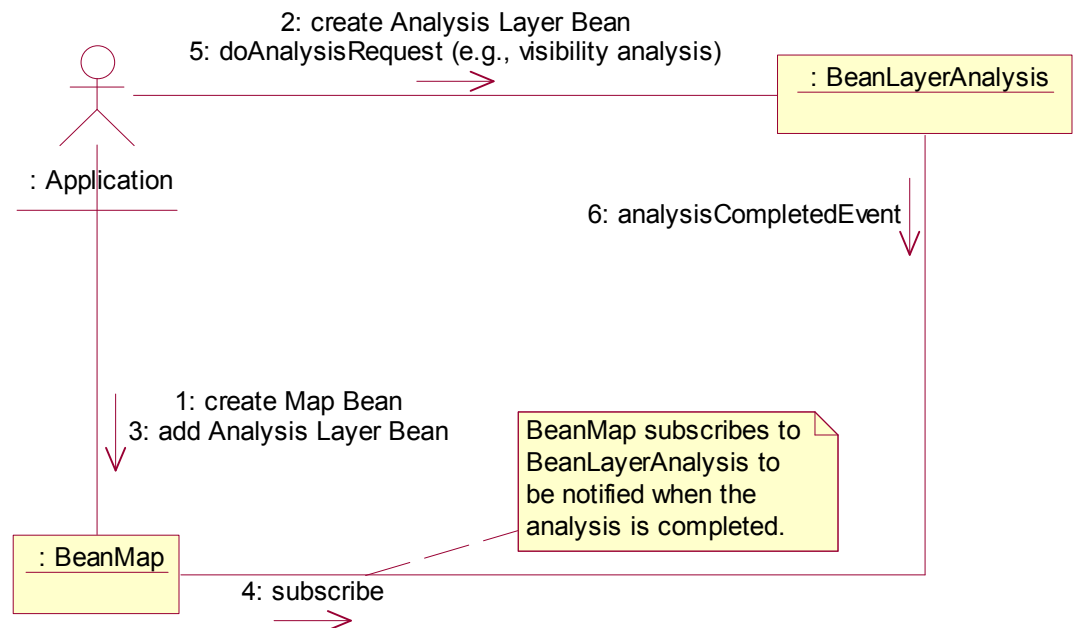
- d. *BeanLayerRaster*: Describes functionalities of the “raster layer” component in this framework. It implements “InterfaceLayerRaster” interface that must be implemented by beans written as Raster Layer Bean in this framework. Raster map management is done in this bean. Raster Layer Bean manages showing raster layers according to the scale of the map of which paths are included in the configuration file. For example, if the scale of the map is 375000, raster maps with scale of 1/500000 are shown. Beans that want to

listen to Raster Layer Bean implement “ListenerLayerRaster” interface to be notified with the events occurred in this bean. Raster Layer Bean is a listener of Map Bean in this framework. When Raster Layer Bean is added to Map Bean, it registers itself with Map Bean and Map Bean also registers itself with Raster Layer Bean automatically. Raster Layer Bean notifies its listeners, when visibility of raster map is set from outside, and raster maps are loaded according to the new scale. Figure 3.7 illustrates the registration mechanism of the Raster Layer Bean with the Map Bean that is the core bean in this framework.



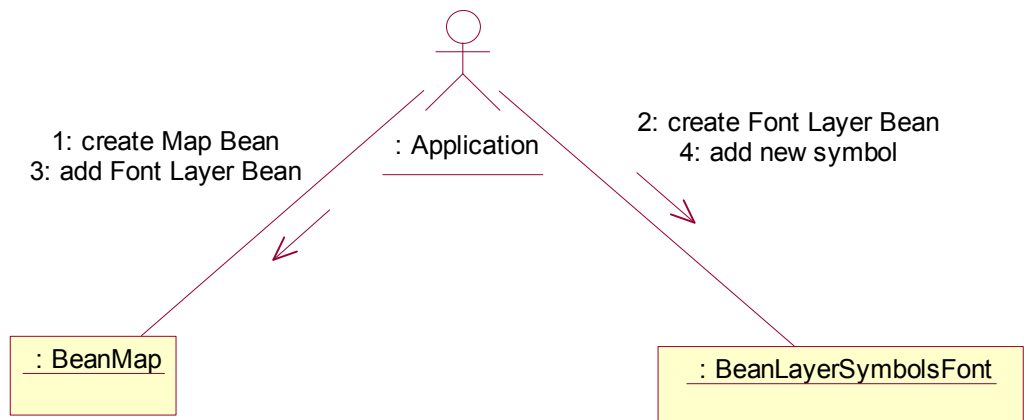
**Figure 3.7** Communication between the BeanMap and the BeanLayerRaster components in the GIS Domain Framework

- e. *BeanLayerAnalysis*: Describes functionalities of the “analysis layer” component in this framework. It implements “InterfaceLayerAnalysis” interface that must be implemented by bean written as Analysis Layer Bean in this framework. Beans that want to listen to Analysis Layer Bean implement “ListenerLayerAnalysis” interface to be notified with the events occurred in the bean. When Analysis Layer Bean is added to Map Bean, Map Bean registers itself with Analysis Layer Bean automatically. Analysis Layer Bean notifies its listeners when a pixel coordinate is selected by a mouse click in the outside world to request conversion of these pixel coordinates to world coordinates. Figure 3.8 illustrates the registration mechanism of the Analysis Layer Bean with the Map Bean that is the core bean in this framework.



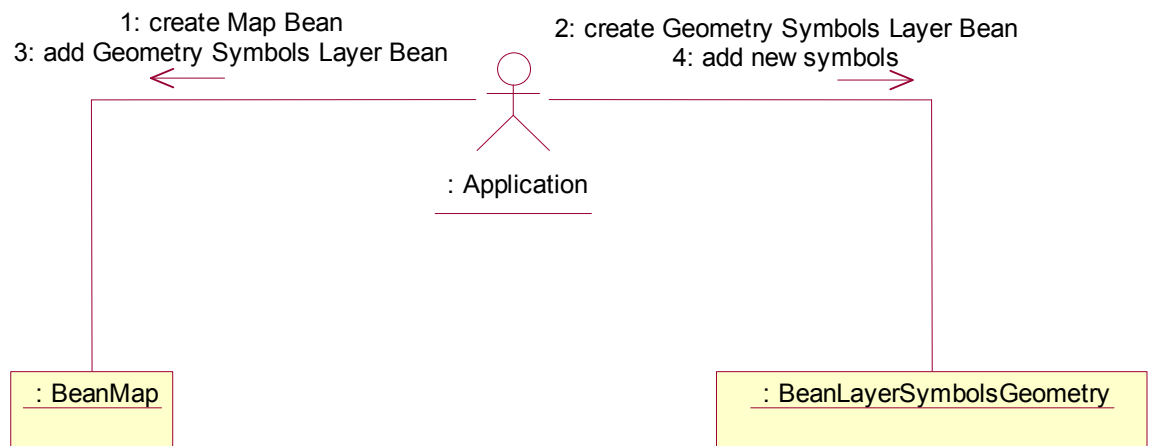
**Figure 3.8** Communication between the BeanMap and the BeanLayerAnalysis components in the GIS Domain Framework

f. *BeanLayerSymbolsFont*: Describes functionalities of the “font symbols layer” component in this framework. It implements “InterfaceLayerSymbolsFont” interface that must be implemented by the Font Symbols Layer Bean in this framework. Figure 3.9 illustrates the registration mechanism of the Font Symbols Layer Bean with the Map Bean that is the core bean in this framework.



**Figure 3.9** Communication between the BeanMap and the BeanLayerSymbolsFont components in the GIS Domain Framework

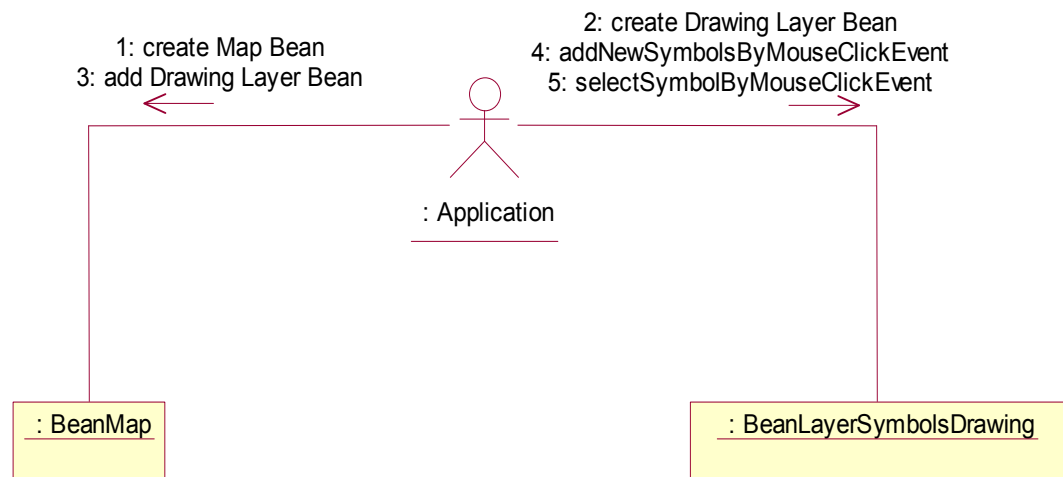
g. *BeanLayerSymbolsGeometry*: Describes functionalities of the “geometry layer” component in this framework. It implements “InterfaceLayerSymbolsGeometry” interface that must be implemented by beans written as Geometry Symbols Layer Bean in this framework. Figure 3.10 illustrates the registration mechanism of the Geometry Symbols Layer Bean with the Map Bean that is the core bean in this framework.



**Figure 3.10** Communication between the BeanMap and the BeanLayerSymbolsGeometry components in the GIS Domain Framework

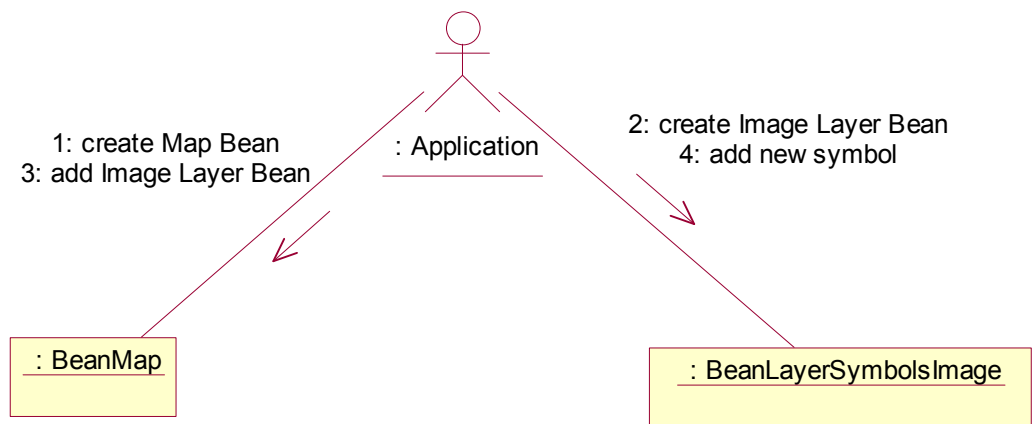
*h. BeanLayerSymbolsDrawing:* Describes functionalities of the “drawing layer” component in this framework. It implements “InterfaceLayerSymbolsDrawing” interface that must be implemented by the Drawing Symbols Layer Bean in this framework. Figure 3.11 illustrates the registration mechanism of the Drawing Symbols Layer Bean with the Map Bean that is the core bean in this framework.





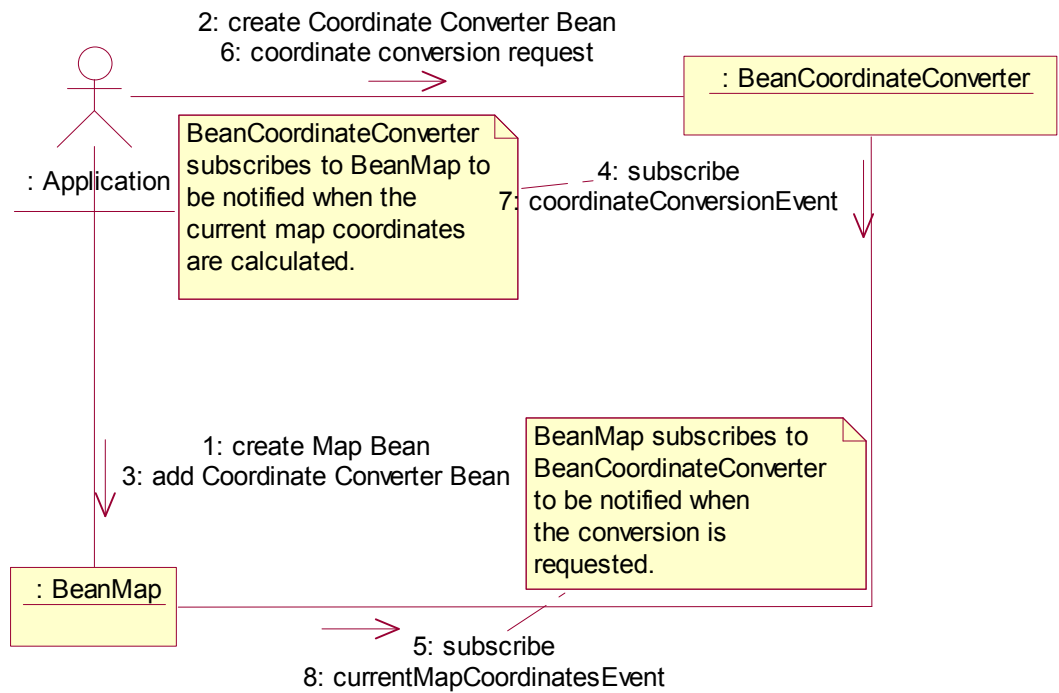
**Figure 3.11** Communication between the BeanMap and the BeanLayerSymbolsDrawing components in the GIS Domain Framework

- i. *BeanLayerSymbolsImage*: Describes functionalities of the “image symbols layer” component in this framework. It implements the “InterfaceLayerSymbolsFont” interface that must be implemented by the Image Symbols Layer Bean in this framework. Figure 3.12 illustrates the registration mechanism of the Image Symbols Layer Bean with the Map Bean that is the core bean in this framework.



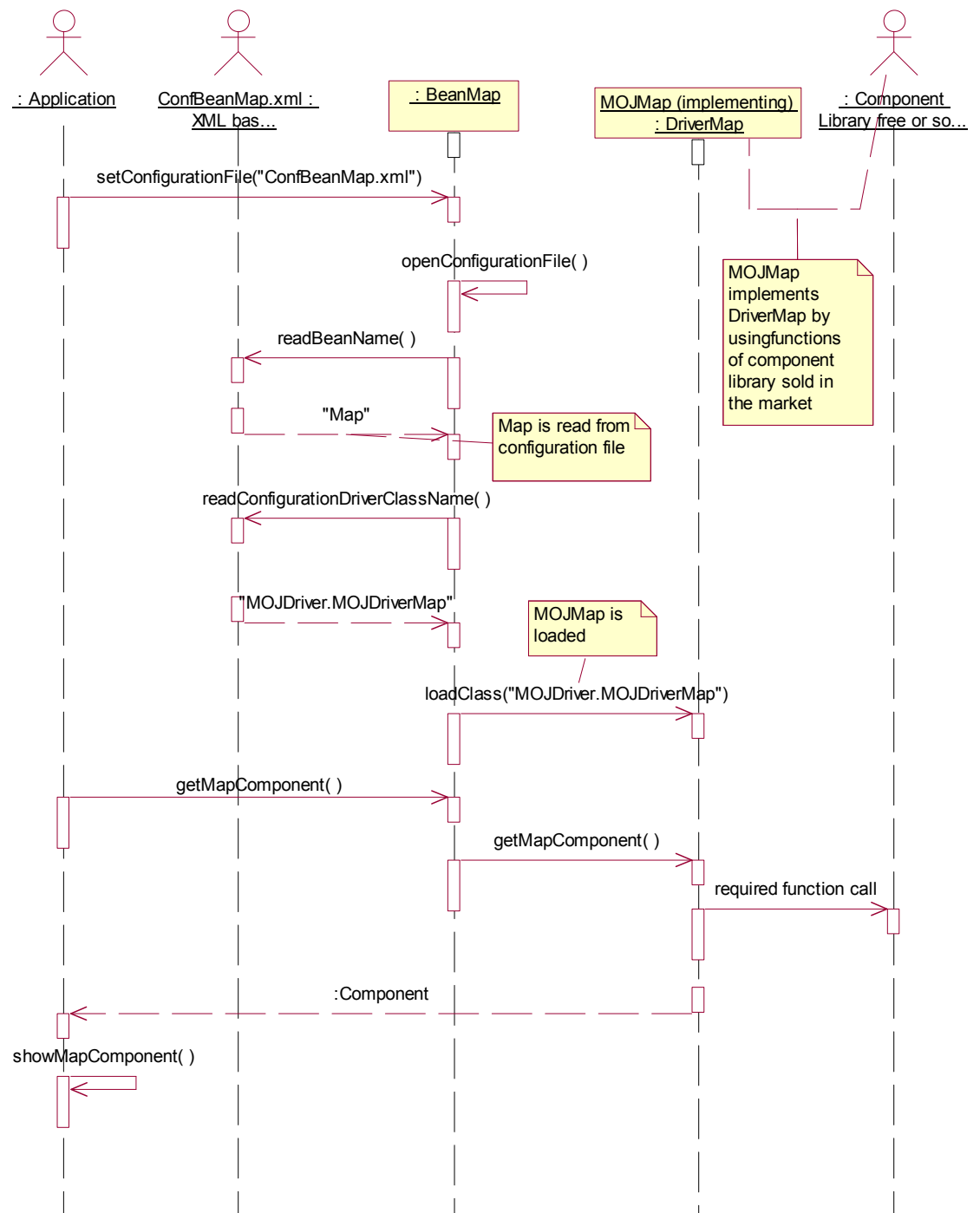
**Figure 3.12** Communication between the BeanMap and the BeanLayerSymbolsImage components in the GIS Domain Framework

- j. *BeanTrackingFontLayer, BeanTrackingGeometriesLayer, and BeanTrackingImagesLayer*: These components are the tracking forms of the *BeanLayerSymbolsFont, BeanLayerSymbolsGeometry, and the BeanLayerSymbolsImage* beans.
- k. *BeanCoordinateConverter*: Describes functionalities of the coordinate converter component in this framework. It implements the “InterfaceCoordinateConverter” interface that must be implemented the Coordinate Converter Bean in this framework. Beans that want to listen to Coordinate Converter Bean, implement “ListenerCoordinateConverter” interface to be notified with the events occurred in this bean. The Coordinate Converter Bean is a listener of the Map Bean in this framework. When the Coordinate Converter Bean is added to the Map Bean, it registers itself with the Map Bean and the Map Bean also registers itself with the Coordinate Converter Bean automatically. The Coordinate Converter Bean notifies its listeners when a request arrives for conversion of pixel coordinates of a selected point. Figure 3.13 illustrates the registration mechanism of the Coordinate Converter Bean with the Map Bean that is the core bean in this framework.



**Figure 3.13** Communication between the BeanMap and the BeanCoordinateConverter components in the GIS Domain Framework

Component libraries that are currently being used in the projects and other component libraries in the market were examined. It was found that Map Objects Java (MOJ) from ESRI meets most of the requirements in the applications. Therefore, most of the requirements in the domain will be covered by implementing MOJ software. In addition to MOJ, the codes that were written previously have been collected, converted into components and added to the component library. The codes developed in other languages have been re-written in Java. New components were developed for the requirements that are not covered by any component. Figure 3.14 illustrates that BeanMap uses “ConfBeanMap.xml” configuration file to read the name of the driver class implemented by MOJ and called “MOJMap”. Application sets configuration file for BeanMap and uses this bean.



**Figure 3.14** An example of using configuration file with the BeanMap component

To show the flexibility in using component library, two component libraries commercially available have been used. One of them was MOJ and the other was OpenMap that is an opensource library. In addition, to validate changeability for usage of component library mechanism of the framework, two driver classes have been implemented by using OpenMap. The configuration files have been filled for these two drivers. Different applications have been run that utilize the component library commercially available, MOJ and OpenMap without modifying any code but only adapting the configuration files for the components. Furthermore, to test the changibility at runtime, application can modify the component library by changing configuration file. Two configuration files are shown below for using MOJ and OpenMap. “MOJDriver.MOJDriverMap” and “OMDriver.OMDriverMap” classes implement the DriverMap interface in the framework.

```
<ConfigurationFile>
  <BeanCallDefinitions>
    <BeanName>MOJMap</BeanName>
    <BeanDriverClass>MOJDriver.MOJDriverMap</BeanDriverClass>
  </BeanCallDefinitions>
</ConfigurationFile>
```

```
<ConfigurationFile>
  <BeanCallDefinitions>
    <BeanName>OMMap</BeanName>
    <BeanDriverClass>OMDriver.OMDriverMap</BeanDriverClass>
  </BeanCallDefinitions>
</ConfigurationFile>
```

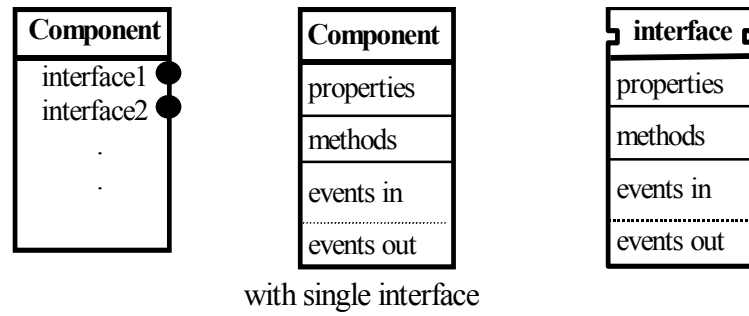
## CHAPTER 4

### INTERFACE LAYER TO UTILIZE JAR LIBRARIES AS COMPONENTS

#### 4.1. Importing Beans into COSECASE Tool and System Design

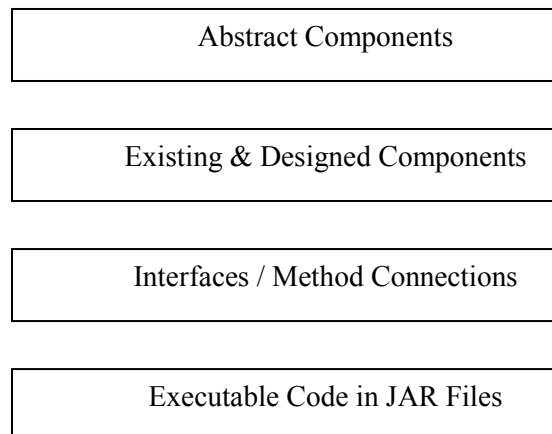
An *Application Engineering* process develops software products from software assets created by a domain engineering process. This chapter of the thesis presents the interface layer that has been developed adapting external components to a design tool. This interface layer is used to access Java Archive (JAR) libraries that include Java Beans. JAR libraries are hence made available for use in the design tool as components, extending the tool for development beyond design. These executable components are used in designing applications with the Component Oriented Software Engineering (COSE) approach. In addition, components mentioned in the previous chapter have been imported into the COSECASE and used to design and generate an application.

The component-oriented modeling tool (COSECASE) has been developed to design a system with respect to the COSE approach [8]. System is designed by deploying copies of the icons on the toolbar. The component icon symbolizes existing components. Each component has interfaces to represent its properties, methods, and events as shown in Figure 4.1. An Interface is the connection point of a component; services requested from a component are invoked through this interface. Using the tool, a designer places component and interface icons on the drawing panel. A properties panel is opened for an icon by clicking the right mouse button on the icon. By using the properties panel, name of the component, its interfaces, properties, methods, and in/out events are entered. Using the previous version of the tool, system design had to be carried out manually.



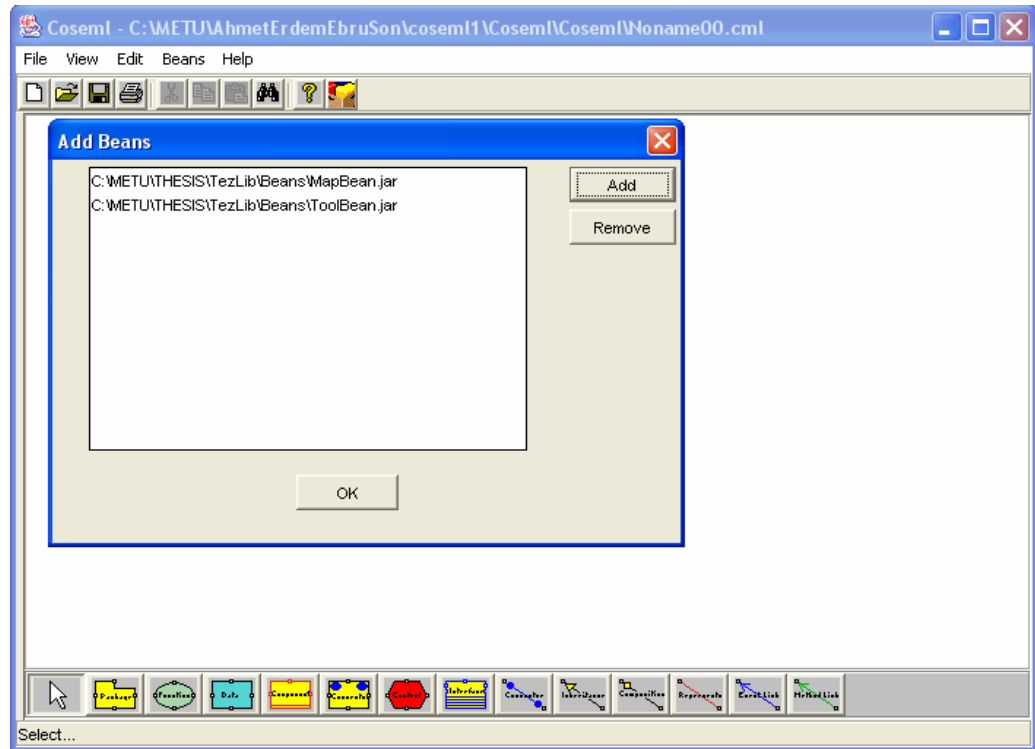
**Figure 4.1** Component and Interface Symbols in COSEML

After the enhancement, COSECASE can be used as a GIS framework. Now it is possible to model the decomposition of logical-level components. Then, existing components and new components can be linked to those abstract components. Finally, method-level connections can be defined using the interfaces. Figure 4.2 shows the layers in the enhanced tool.



**Figure 4.2** Conceptual Layers in the Framework

To design a system using the COSECASE tool with increased leverage, existing executable components must be used. These components must be imported into the tool. Their properties, methods and events must be shown when these components are carried to the workspace. The interface layer developed for the thesis work assumes that the JAR libraries include Java Beans and beaninfo classes. A menu operation that is used to import JAR files into the tool is placed in the COSECASE. After selecting JAR files by using this menu option, bean-info of the beans in the JAR file are read. A screen from the COSECASE for this operation is shown in Figure 4.3:

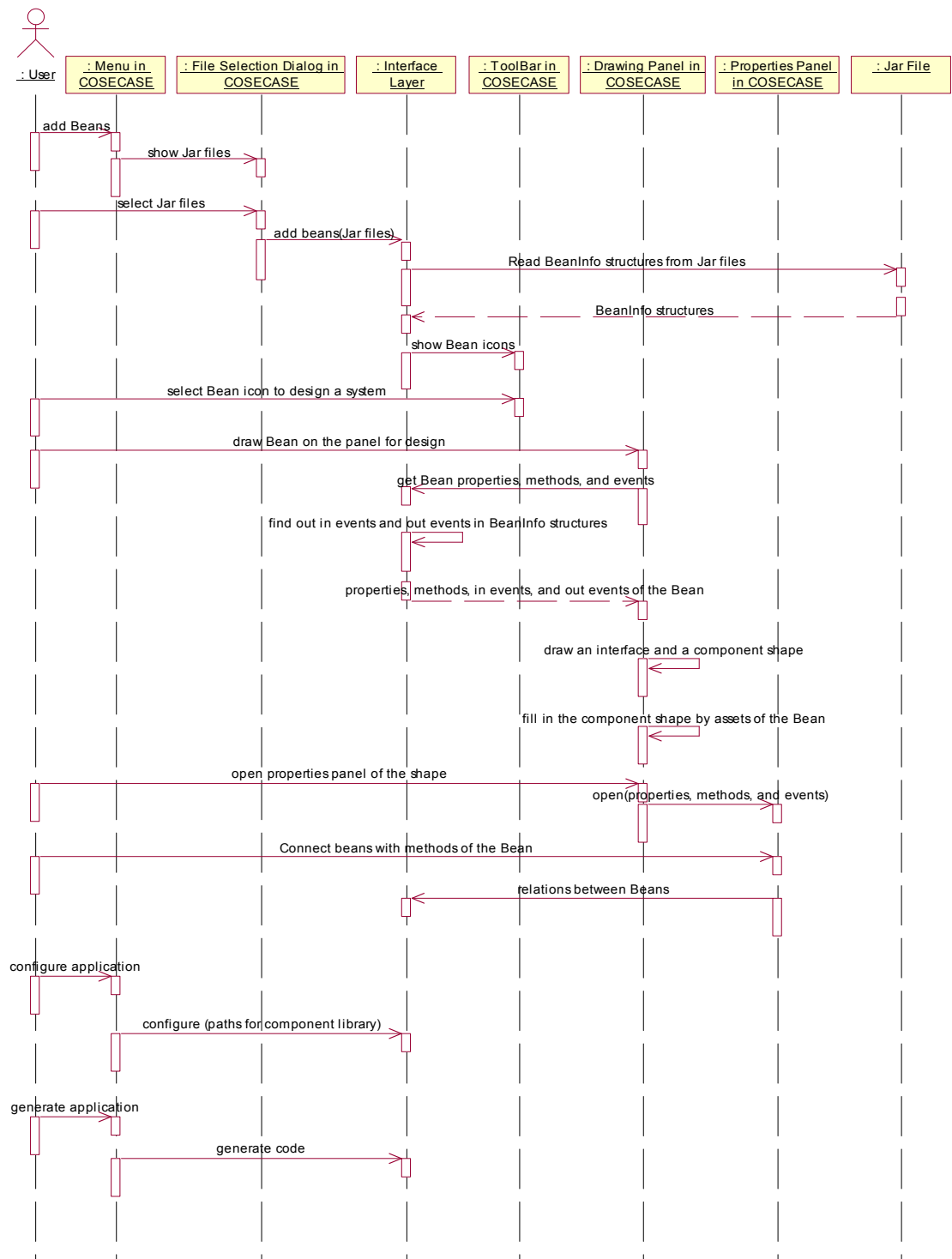


**Figure 4.3** Importing existing bean components into COSECASE

The name of the bean, properties, methods, and in/out events of the component are obtained from the bean-info structure. In addition, the icon for the Java Bean is

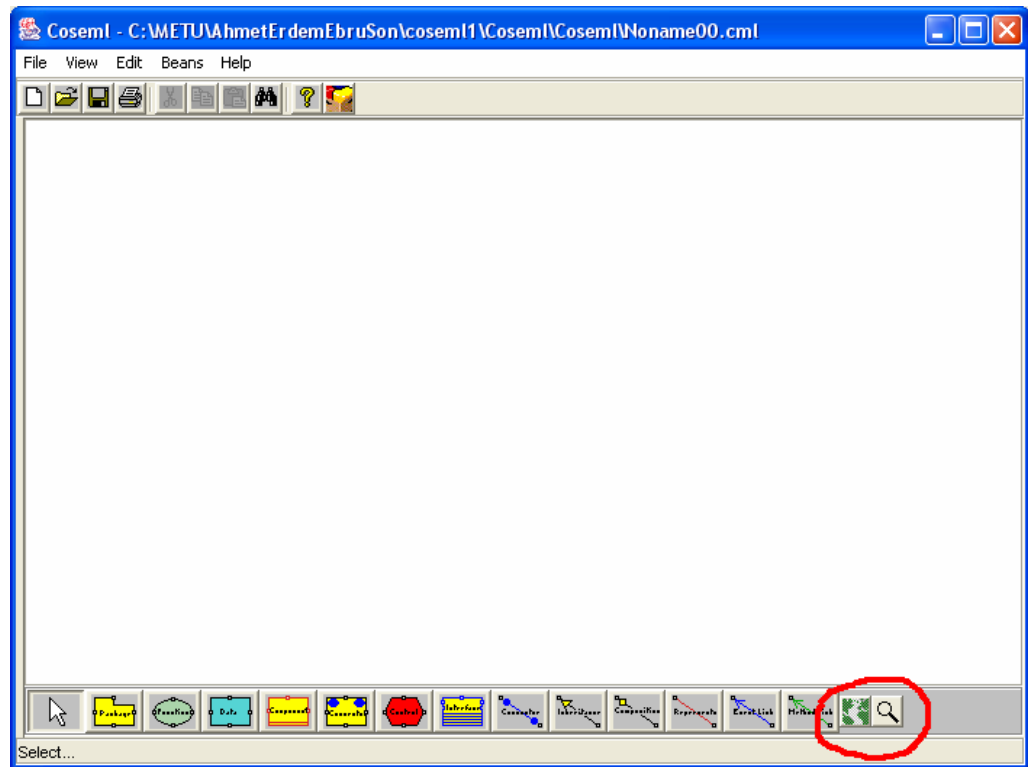


retrieved from the bean-info, icon file is copied from the JAR library, and the icon is shown on the toolbar of COSECASE. In Figure 4.4, a sequence diagram illustrates usage of these JAR files as components in COSECASE:



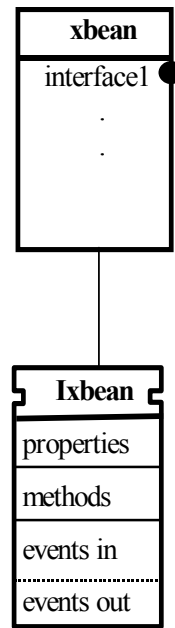
**Figure 4.4** Using JAR files as components in the COSECASE

Bean icons represent the imported executable components on the toolbar. By pressing over a bean icon and then dragging the component to the drawing panel of the tool, the component shape representing the bean is shown. Figure 4.5 shows the screen from the COSECASE tool for this operation:



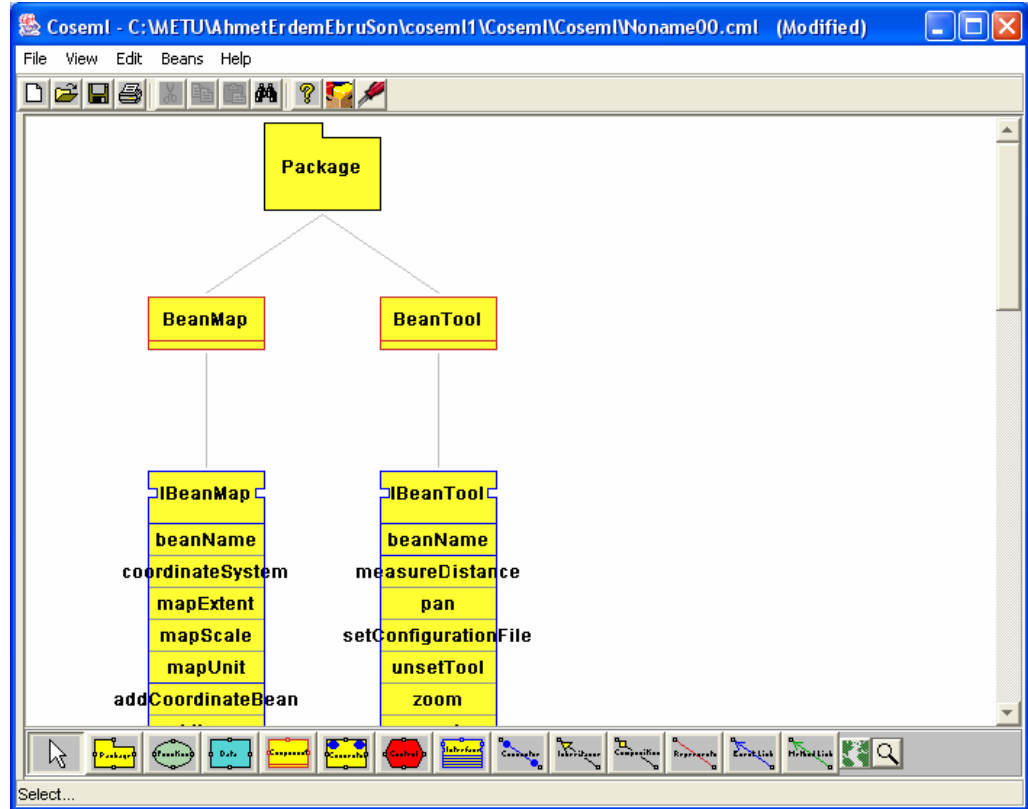
**Figure 4.5** Icons of Imported Bean Component in COSECASE

A component and an interface for that component are drawn on the panel. This interface presents the name of the bean, as read from the bean-info. Properties, methods, image, and in/out events of the component retrieved from the bean info are shown on the interface of the component. For example, if xbean is included in the JAR library, the corresponding component and the interface are shown in Figure 4.6:



**Figure 4.6** Bean Components from JAR libraries in COSECASE

A system can be designed by using these imported executable components. System is decomposed into smaller parts. When the parts are small enough to correspond to the imported components, they are selected and placed in the system design. A simple example is shown in Figure 4.7. The system consists of the BeanMap and the BeanTool components in the figure.



**Figure 4.7** Beans in COSECASE

The Java Beans bean-info structure includes information about properties, methods, parameter types of the methods, bean image, and events that are fired by the bean. The Reflection package in Java is used to load a class at runtime. Name of the bean class is retrieved at runtime. While importing a bean in a JAR file into COSECASE, the bean and bean-info classes are loaded by using the reflection package, since information about the bean class is not known statically. Therefore, classes are loaded dynamically. Properties, methods, image and parameter types are obtained from the beaninfo and names of these items are shown on the panel. The sequence for loading bean components dynamically is presented below:

```
theBeanClasses [classNo] = Class.forName("theBeanName");
```

```
theBeans [classNo] = theBeanClasses[classNo].newInstance();
```

```

theBeanInfoClass [classNo] = theBeanName.concat("BeanInfo");

theBeanInfos [classNo]=Introspector.getBeanInfo(theBeanClasses[classNo]);

Image beanImage = theBeanInfos [classNo].getIcon (16);

properties = theBeanInfos[classNo].getPropertyDescriptors();

methods = theBeanInfos[classNo].getMethodDescriptors();

```

In COSEML and COSECASE, a component has two types of events: events that component receives and events that component fires. On the other hand, bean-info includes information about bean events that are fired by the bean. This structure corresponds to “out-events” in COSEML. Therefore, to find “in-events” that a Java Bean gets, the developed interface layer searches all the listener types that the bean implements and then gets the methods of those listener interfaces by using the reflection package again.

There are two ways for locating the “out-events” for the bean. One of them is getting events fired from the bean-info by using the Introspector package of Java:

```

eventsSet = theBeanInfos[classNo].getEventSetDescriptors();

```

The other is using listeners as mentioned above. This process is conducted by using naming conventions in Java. For example, to add a listener called XYZ, the bean includes `addXYZListener(XYZ)` method. This interface layer searches all `add<ListenerType>Listener` methods in the bean and gets the methods for all the listener types that can be added to the bean. In this way, methods that are used to add listeners to the bean are searched and listeners that can be added to the bean are obtained. The Listener class is loaded by the reflection utility and the methods of the listener are accessed. The methods of these listeners are “out-events” for the bean, since those methods are called from the bean. This way, gathering “in-events” and “out-events” becomes standardized way. The pseudocode describing this operation is shown below:

```

for all methods {
    if (method name.startsWith("add") && method name.endsWith("Listener")) {
        get parameter types of the method as listener class type
    }
    for all retrieved parameter types {
        load listener class
        get methods of listener class
        hold names of these methods as out-event names
    }
}

```

It is assumed that all listeners implement the "java.util.EventListener" interface. To find out events that a bean receives, all the interfaces that bean implements are searched. If an interface implements "java.util.EventListener", it means that the interface is a listener. Therefore, the methods of this listener are “in-events” for the bean, since the bean implements this interface. The pseudocode describing this operation is shown below:

```

for all interface classes that a bean implements
    call getInEvents function for the interface

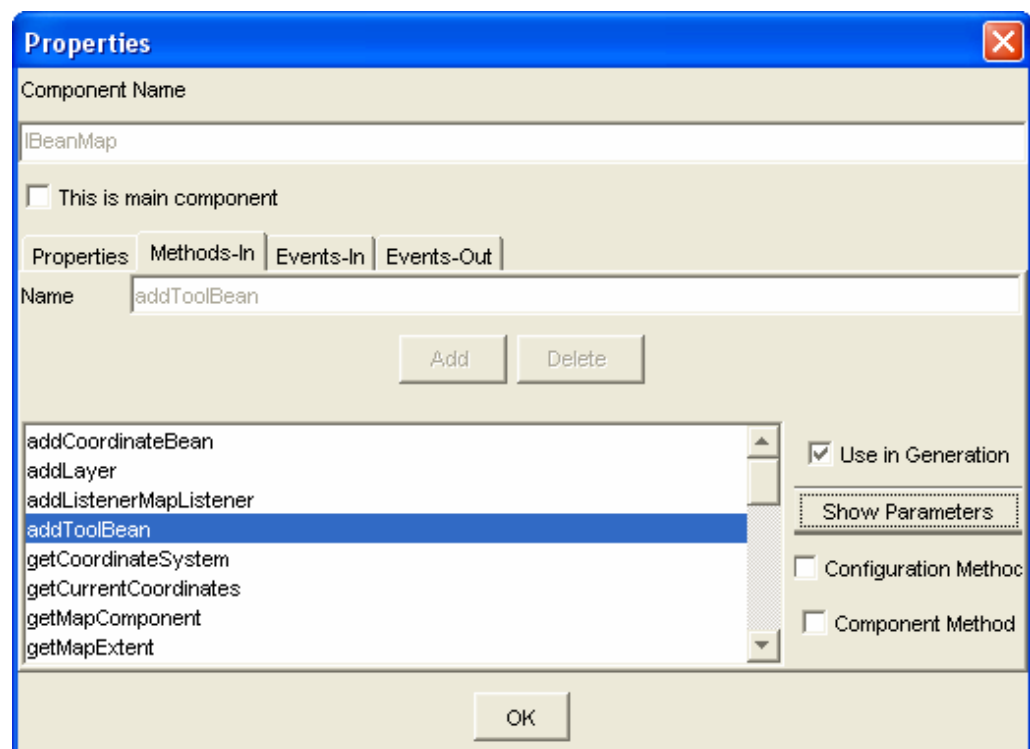
// _____

public void getInEvents(Class _interfaceClass, java.util.Vector methInNames) {
    if this interface is not "java.util.EventListener"
        for all interfaces that this interface implements
            call getInEvents function for the interface recursively
        get the methods of the interfaces as in-events
}

```

All the interfaces of the bean are searched recursively in the “getInEvents” function until "java.util.EventListener" is reached. When it is reached, names of the methods in the listeners are marked as “in-events”.

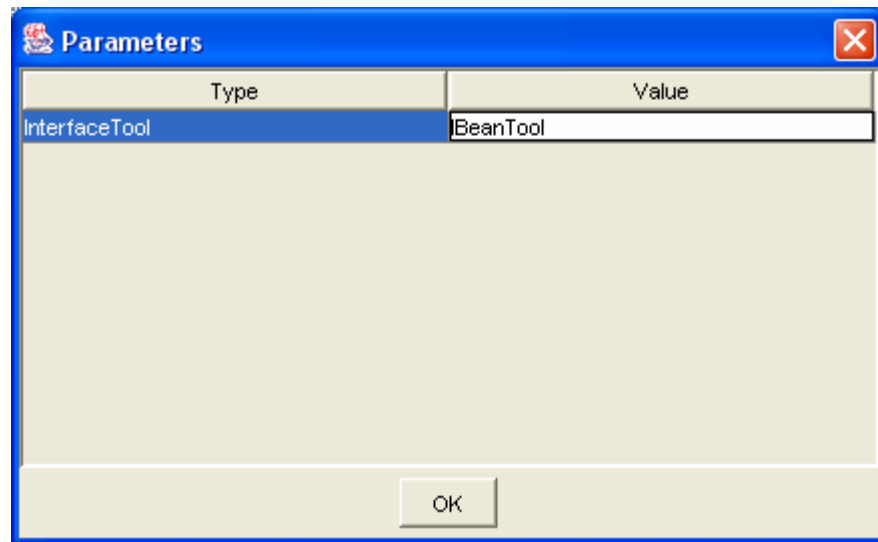
In the previous version of COSECASE, properties of the components are shown on the properties dialog by clicking the right button of the mouse on the component icon. This dialog box shows the name, properties, methods, and the in/out events for the component. The current version of the tool automatically displays such values when existing imported component is used, since these values are read from the bean-info record. Figure 4.8 shows the properties panel for BeanMap bean in COSECASE. “Use In Generation” check box is marked when the codes for the selected method be generated. “Configuration Method” check box is marked when the selected method will be used for setting configuration file.



**Figure 4.8** Properties Dialog Box in COSECASE



Parameter types of the methods are also read from the bean-info record. By using this utility of the bean-info, another dialog box showing types of the parameters are shown for the methods. In this dialog box, user can enter values for the parameters of which types are shown. Unfortunately, there is no way to give an appropriate name to the parameters from the bean-info. Therefore, only the types of the parameters are shown on the dialog box. The values entered in this dialog box are used in code generation. Figure 4.9 shows the Parameters Dialog Box:

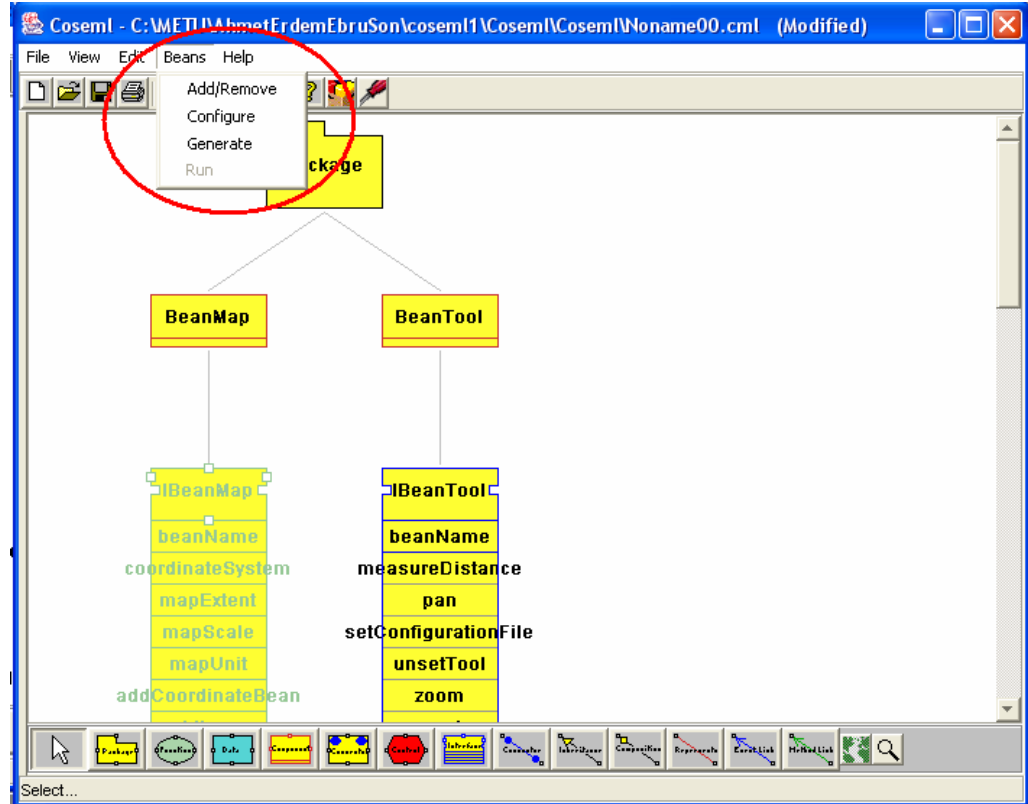


**Figure 4.9** Parameters Dialog Box in COSECASE

Semi-automated code generation and application execution utilities are also added to COSECASE. By using the information in the properties dialog box, code generation is done by the tool. Users select methods that will be used in code generation by checking the “Use In Generation” checkbox. Figure 4.9 shows this condition. If a method is signed to be used in code generation, parameters dialog box shown in Figure 4.9 is opened and the user is requested to enter values for the parameters. While generating the code, tool looks for the signature of the method. If the method is signed for code generation, the code is generated for the method by

using the values entered in the parameters dialog box. Generated code is written in a file named as “Application1.java” in the COSECASE directory. This file is a simple Java source file that includes the frame panel class. This Java file is compiled during the generation time and packaged in a JAR file. A “bat file” is also generated in the generation time by the tool. This file includes necessary information to run the generated code. Application generation is done by running “Runtime.getRuntime().exec(cmd)” command from the tool. Sometimes, this utility does not work because of the deficiency of Java in this command.

In the previous section, a GIS domain framework is represented and components of this framework are packaged as JAR libraries. A GIS application was designed by using COSECASE tool and existing components in the domain by importing these JAR libraries into COSECASE in this thesis. Figure 4.10 illustrates this simple application design in COSECASE. The icons on the toolbar represent the bean components. A GIS application is designed by clicking on these component icons and dragging these components on the panel. The executable code is provided by the JAR libraries. A configuration part is added to the menu, since this domain framework uses component library commercially available. Path of the component library is configured from this menu. When the “generate button” is pressed, executable code is generated for the application by using the parameters entered on the property dialog. When “run application button” is pressed, the application is executed.



**Figure 4.10** Beans Menu in COSECASE

## 4.2. Experimental Results

Three simple applications were generated by using the interface layer and also by coding them manually for experimental purposes. One of the applications had 2, another had 3, and the last one had 4 components, which were implemented as a result of Domain Implementation phase as described in Chapter 3. Every application was generated more than one time through coding and also by integration through the developed framework. Average generation times in minutes, sizes of the applications in kilobytes (KB), and the number of lines in the generated applications were measured for each case. The experimental efficiency metrics are reported as KB per minutes and lines per minutes. The results of using the interface layer are given in Table 2, and the results corresponding to coding the application manually are given in Table 3.

**Table 2** Results of the experiments by using the interface layer

Number of components used in the applications	COSECASE with the interface layer				
	Time (min)	Size (KB)	Lines of Code	Average	
				KB / min	Lines / min
2	3	2.7	57	0.9	19
3	3.7	3.1	63	0.8	17
4	4	3.3	67	0.8	16.8

**Table 3** Results of the experiments by coding the application.

Number of components used in the applications	Coding the application manually				
	Time (min)	Size (KB)	Lines of Code	Average	
				KB / min	Lines / min
2	6	2.8	56	0.5	9.3
3	7	2.9	60	0.4	8.6
4	8	3.3	63	0.4	7.8

The results of using the interface layer and coding the application manually were compared. It was noticed that application generation by using the interface layer is 2 times faster and more efficient than coding the application manually. The comparison results are given in Table 4.

**Table 4** Comparison of the values of interface layer / coding manually

Number of components used in the applications	KB/min values: through framework/ through coding	lines/min values through framework/ through coding
2	1.8	2
3	2	2
4	2	2.1

Besides these experimental results, using the interface layer has some advantages experimented while generating those simple applications:

1. One selection, one mouse-click, or one input is used to generate one line of code in the interface layer. On the other hand, this line of code needs to be entered manually for coding the application.
2. Since the imported components are used through the interface layer, the names of the components are retrieved from the source code of

these components. Therefore, the generated code becomes more standard in comparison to the coding the application alternative. Even for the case where different persons generate the same design at different times; the generated code remains similar.

3. It was noticed that more errors were produced while coding the application manually. In addition, finding the error and correcting it takes more time.
4. With the interface layer the design and the code of the application are kept in the same environment. Therefore, changing the code of the application means changing some parameters or settings just in the design.

### **4.3. Evaluation of Other Related Work**

Software development based on frameworks including pre-fabricated components is a way to improve productivity in Software Engineering. Developing software by composing the components in a framework and generating applications automatically is an important issue. A lot of efforts have been spent on this issue.

Different approaches are introduced in [31], [32], [33]. In [31] components from different sources can be imported into the framework. An eXtended Markup Language (XML)-based file defines the construction of the component. These components are re-constructed as JavaBeans in the framework by manipulating this XML-based file. Composition of components is handled in a Graphical User Interface (GUI)-based environment by using the properties of the JavaBeans manually. Relations defined graphically are inserted into an XML-based file in generation time. Java source code is generated from this file. This approach supports the using of the executable components from different sources with the help of XML-based file in the system design. Whereas this approach requires knowledge about the properties, methods, and events of the components to import them into the framework, we proposed using of JAR files including JavaBeans and bean info structures in our framework. Using the beaninfo structures in our approach directly generates Java source code. Although there is no need to know the assets of the components to import

them to the framework, only JAR files including JavaBeans can be imported to the framework. This is a disadvantage in comparison to the work described in [31].

To compose the components some structural relations including plugs are defined in [32]. A structural relation consists of two or more plugs containing a set of interfaces. Components are attached to these plugs by implementing the interfaces that is represented by the plugs. Components are composed in a GUI-based environment again. Pre-defined structural relations bring some overhead to the composition of two components. All the three studies including our study above are using a GUI-based environment to compose components. Therefore, code is generated semi-automatically since components are integrated manually. Full-automated code generation is still an open point in our study.

In [33], a new algorithmic approach is introduced to integrate components. A component is modeled with data, function and, control elements. The control elements define the interaction of the services between the components. The components are integrated by using the adapter components that solves the mismatches between the integrating parts. Automatic and dynamic generation of the adapter components at runtime makes the approach valuable. Also a meta-semantic language is introduced to define the components and the integration of them. These specifications are then translated into the generation of the adapters. Integrating components and full automatic code generation is thought as a future work in our study.

As a summary, it can be said that our approach is somewhat comparable to similar attempts. We found ours to be more practical especially in the applied GIS domain. Less code writing and less information requirement on the definition of components is favored over the disadvantage of the limited input format that is jar files.

## **CHAPTER 5**

### **CONCLUSION**

An interface layer has been developed that uses Java Archive (JAR) libraries to locate deployed components and imports them into the COSECASE tool. This interface layer retrieves properties, methods, and in/out events of these components from the existing codes. Such elements are re-organized as COSEML structures that are component interfaces. Organizations deploy their Java Bean components as JAR libraries. Therefore, systems can be developed by using executable components in COSECASE in the form of JAR libraries. In addition, semi-automated code generation and application execution is enabled by selecting methods of the components that will be used in code generation and entering values for the parameters of the methods.

The cornerstone of the enhanced tool is using deployed components in system development, code generation for the application, and execution of the application within COSECASE. There is no need to enter properties, methods, in/out events for the existing components in COSECASE, since these items are obtained from the components automatically. Components are imported from JAR libraries and dragged into the drawing panel of the tool for composing systems.

#### **5.1. Work Conducted**

In the thesis, requirements for the GIS domain have been collected from the product line of the company for domain engineering process. Existing domain knowledge and existing codes have been collected from the experts. Domain has been analyzed, boundaries of the domain have been determined and domain has been simply modeled by using Feature Oriented Domain Analysis (FODA). Domain Architecture has been modeled with Unified Modeling Language (UML) and components have been developed. Components were developed in three ways:



- 1- Existing Java codes in the organization have been converted to components directly.
- 2- Existing codes in other languages were first converted to Java and then converted to components.
- 3- Components were developed from scratch, if there was no existing code for the corresponding requirement.

These components were developed as beans. Bean-info classes for the beans have also been written. Java Bean components have been packaged into “JAR files” for deployment. For the second part of the thesis, COSECASE has been debugged. Errors in the tool have been corrected. An interface layer has been developed for the tool to import JAR libraries and to use these libraries as components. Beans contained in JAR libraries have been integrated with the component mechanism in COSECASE. A new mechanism has been added to the tool to show parameter types for the parameters of the methods for these bean components. Semi-automated code generation and application execution mechanisms have been added to the tool.

## **5.2. Comments**

Although an example application has been generated for the thesis work, no real experiments have been conducted in the industry yet. Also, to assess the added value of this work, a set of three smaller applications were developed and comparisons were conducted by applying alternative approaches on this set. This limited experience, however, was very valuable. Experimental results show that application generation by using the interface layer is 2 times faster and more efficient than coding the application manually. After satisfying the author for the validity of the approach, it was noticed in a defense related engineering organization. Consequently, proposed framework approach is now planned by the organization to be adopted as a method for developing GIS applications. With the developed tool set, it will be possible to achieve a domain engineering framework that will compose components into applications. This goal does not seem to require too much extra effort. A prototype framework has already been developed.

### **5.3. Future Work**

For Java Bean components imported into the COSCASE, one component shape and only one interface shape are currently being displayed on the drawing panel. This interface shape includes information about the services of the component that is recovered from the bean-info. On the other hand, one component can also have more than one interface in COSEML. Therefore, an intelligent mechanism that generates different interfaces from the bean-info can be developed in the future. Semi-automated code generation has been realized in this thesis. Methods that will be used in code generation are selected in the tool manually and then codes for these methods are generated. On the other hand, there is no automated mechanism that finds relations among components. Hence, such a mechanism would enhance the abilities in the integration of components.

## REFERENCES

- [1] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, “Design Pattern Recovery in Architectures for Supporting Product Line Development and Application”, *Modelling Variability for Object-Oriented Product Lines* edited by M. Riebisch, J. O. Coplien, D. Streitferdt (Eds.). BookOnDemand Publ. Co., Norderstedt, pp. 42-57, 2003.
- [2] D.L. Parnas, P.C. Clements, and D.M. Weiss, “Enhancing Reusability with Information Hiding”, *IEEE Tutorial Software Reusability* edited by Peter Freeman, IEEE Computer Society Press, IEEE Catalog #EH0256-8, ISBN 0-8186-0750-5, pp. 83-90, 1987.
- [3] Bertrand Meyer, “Eiffel: Reusability and Reliability”, *IEEE Tutorial Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, IEEE Catalog #EH0278-2, ISBN 0-8186-0846-3, pp. 216-228, 1990.
- [4] Clements Szyperski, “Component Software Beyond Object Oriented Programming”, *Addison-Wesley*, 1998.
- [5] “Software Reuse, Major Issues Need To Be Resolved Before Benefits Can Be Achieved”, *United States General Accounting Office*, January 1993.
- [6] Ruben Prieto Diaz, Gerald A. Jones, “Breathing New Life into Old Software”, *IEEE Tutorial Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, IEEE Catalog #EH0278-2, ISBN 0-8186-0846-3, pp. 152-160, 1990.
- [7] Vedat Bayar, A Process Model for Component Oriented Software Development, *M.S. Thesis*, Middle East Technical University, November 2001.
- [8] Aydın Kara, A Graphical Editor for Component Oriented Modeling, *M.S. Thesis*, Middle East Technical University, April 2001.

- [9] Jim Q. Ning, "A Component Model Proposal", *International Workshop on Component-Based Software Engineering*, pp. 13-16, Los Angeles, CA, USA, 17-18 May 1999.
- [10] Kurt C. Wallnau, "On Software Components and Commercial ("COTS") Software", *International Workshop on Component-Based Software Engineering*, pp. 213-218, Los Angeles, CA, USA, 17-18 May 1999.
- [11] Tricia Oberndorf, Lisa Brownsword, Carol A. Sledge, "An Activity Framework for COTS-Based Systems", *Technical Report, Software Engineering Institute*, October 2000.
- [12] Cecilia Albert, Lisa Brownsword, "Evolutionary Process for Integrating COTS-Based Systems (EPIC): An Overview", *Technical Report, Software Engineering Institute*, July 2002.
- [13] Gail E. Kaiser, David Garlan "Melding Software Systems from Reusable Building Blocks", *IEEE Tutorial Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, IEEE Catalog #EH0278-2, ISBN 0-8186-0846-3, pp. 267-274, 1990.
- [14] Geral Jones, "Methodology/Environment Support for Reusability", *IEEE Tutorial Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, IEEE Catalog #EH0278-2, ISBN 0-8186-0846-3, pp. 190-193, 1990.
- [15] Oh-Cheon Kwon, Seok-Jin Yoon and Gyu-Sang Shin, "Component-Based Development Environment: An Integrated Model of Object-Oriented Techniques and Other Technologies", *International Workshop on Component-Based Software Engineering*, pp. 47-53, Los Angeles, CA, USA, 17-18 May 1999.
- [16] D. Ansorge, K. Bergner, B. Deifel, N. Hawlitzky, C. Maier, B. Paech, A. Rausch, M. Sihling, V. Thurner, S. Vogel, "Managing Componentware Development –Software Reuse and the V-Modell Process", *Proceedings of the 11<sup>th</sup> International Conference on Advanced Information Systems Engineering*, pp 134-148, 14-18 June 1999.

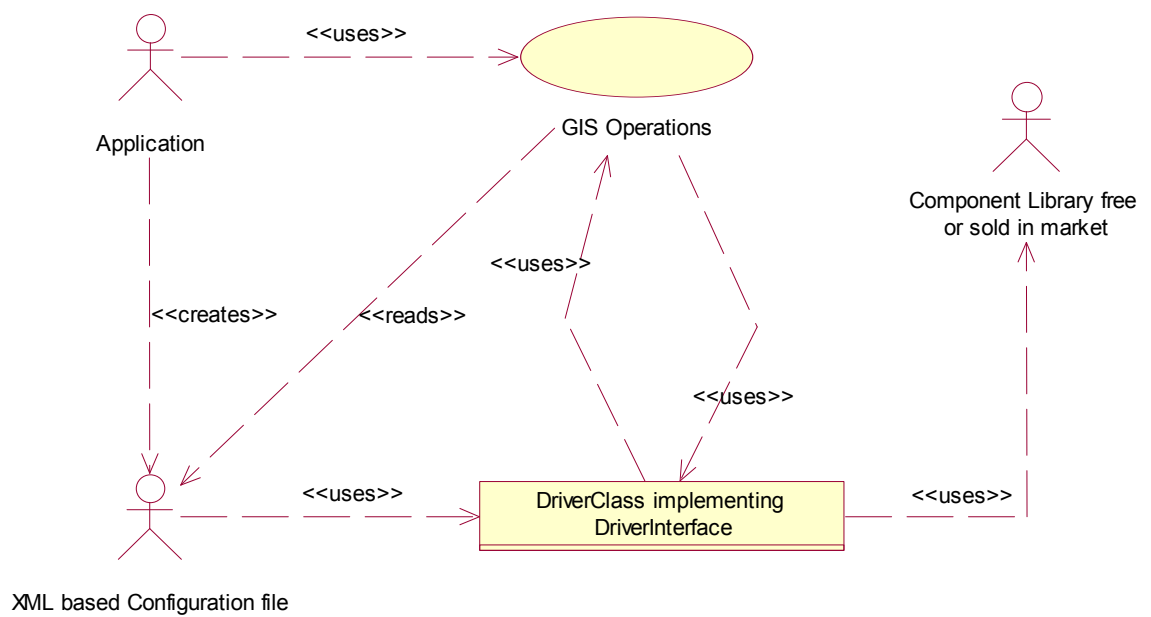
- [17] K. Czarnecki and U. Eisenecker, “Generative Programming: Methods, Techniques, and Applications”, *Addison-Wesley* 1999.
- [18] Rubén Prieto-Díaz, “DOMAIN ANALYSIS: AN INTRODUCTION”, *ACM SIGSOFT Software Engineering Notes*, pp. 47-54, April 1990.
- [19] Rubén Prieto-Díaz, “Domain Analysis for Reusability”, *Proceedings of COMPSAC’87*, pp. 23-29, 1987.
- [20] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, *Technical Report, Software Engineering Institute*, November 1990.
- [21] Sholom G. Cohen, Jay L. Stanley, Jr., A. Spencer Peterson, Robert W. Krut, Jr., “Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain”, *Technical Report, Software Engineering Institute*, June 1992.
- [22] Robert Kurt, Nathan Zalman, “Domain Analysis Workshop Report for the Automated Prompt Response System Domain”, *Technical Report, Software Engineering Institute*, May 1996.
- [23] John D. McGregor, “The Evolution of Product Line Assets”, *Technical Report, Software Engineering Institute*, June 2003.
- [24] Paul C. Clements, Linda M. Northrop, “Salion, Inc.: A Software Product Line Case Study”, *Technical Report, Software Engineering Institute*, November 2002.
- [25] Rosario Girardi and Carla Gomes de Faria, “A Generic Ontology for the Specification of Domain Models”, *Proceedings of 1<sup>st</sup> Workshop Component Engineering Methodology*, pp. 41-50, 24 September 2003, Erfurt, Germany.
- [26] Thomas Cleenewerck, “Component-based DSL Development”, *Proceedings of GPCE03 Conference*, Lecture Notes in Computer Science 2830, pp. 245–264, Springer- Verlag, 2003.
- [27] Carnegie Mellon University Software Engineering Institute (SEI), “Domain Engineering”, [http://www.sei.cmu.edu/domin\\_engineering/domain\\_emg.html](http://www.sei.cmu.edu/domin_engineering/domain_emg.html), March 2005.

- [28] Mark Johnson, “A Walking tour of Java Beans by Mark Johnson”, *Java World* <http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans.html>, February 2005.
- [29] Laurance Vanhelsuwe, “Mastering in Java”, <http://www.javaolympus.com/freebooks/FreeJavaBooks.jsp>, October 2004.
- [30] Esri GIS and Mapping Software, “Geography Matters”, *White Paper* <http://www.esri.com/library/whitepapers/pdfs/geomatte.pdf>, September 2002.
- [31] Vaclav Cechticky, Philippe Chevalley, Alessandro Pasetti, Walter Schaufelberger, “A Generative Approach to Framework Instantiation”, *Proceedings of the 2<sup>nd</sup> International Conference on Generative Programming and Component Engineering*, pp. 267–286, Springer- Verlag, September 2003.
- [32] Theo Dirk Meijler, Serge Demeyer, Robert Engel, “Automated Support for Software Development with Frameworks”, *Proceedings of the 6<sup>th</sup> European Conference held jointly with the 5<sup>th</sup> ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM SIGSOFT Software Engineering Notes, Volume 22 Issue 6, pp. 123–127, November 1997.
- [33] L.K. Jololian, F.J. Kurfess, M.M. Tanik, “Data, Function, And Control As Elements Of Component-Integration”, *Integrated Design and Process Technology*, IDPT-2003, pp. 194–204, June 2003.

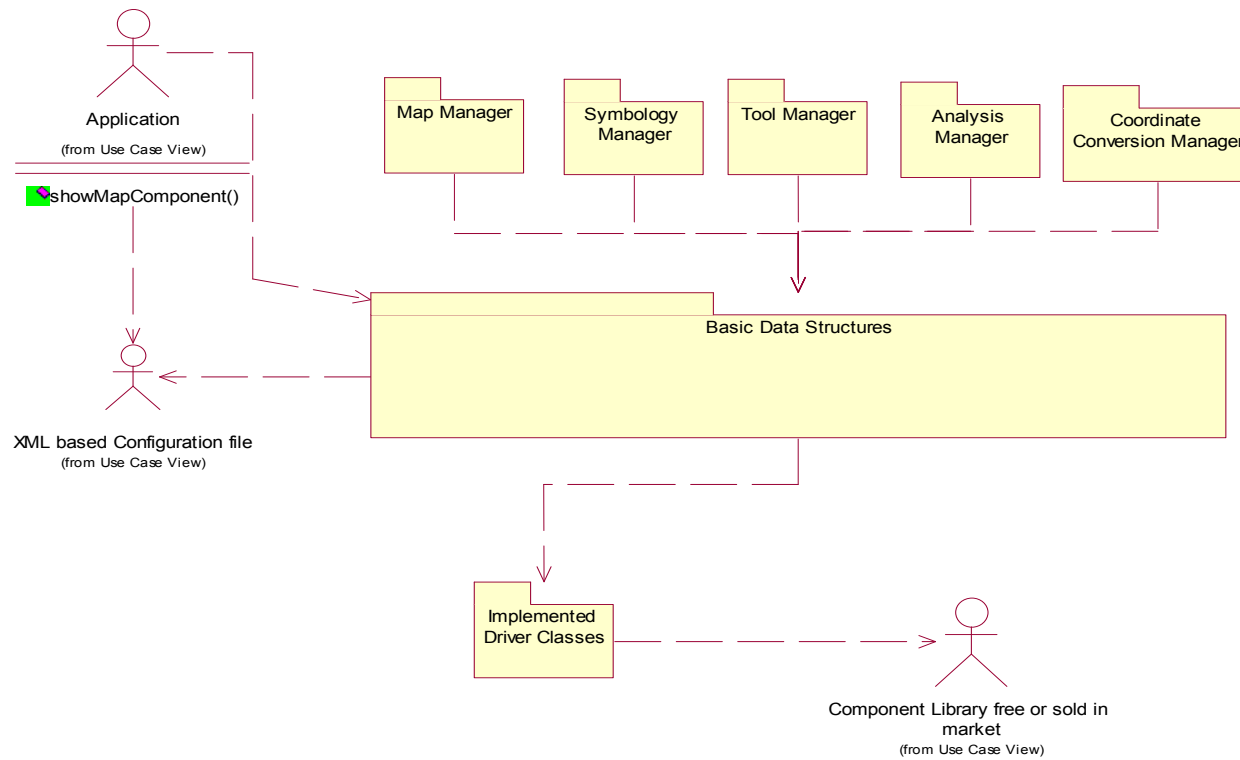
# APPENDIX A

## GIS FRAMEWORK DESIGN

### A.1. Use Case Diagram

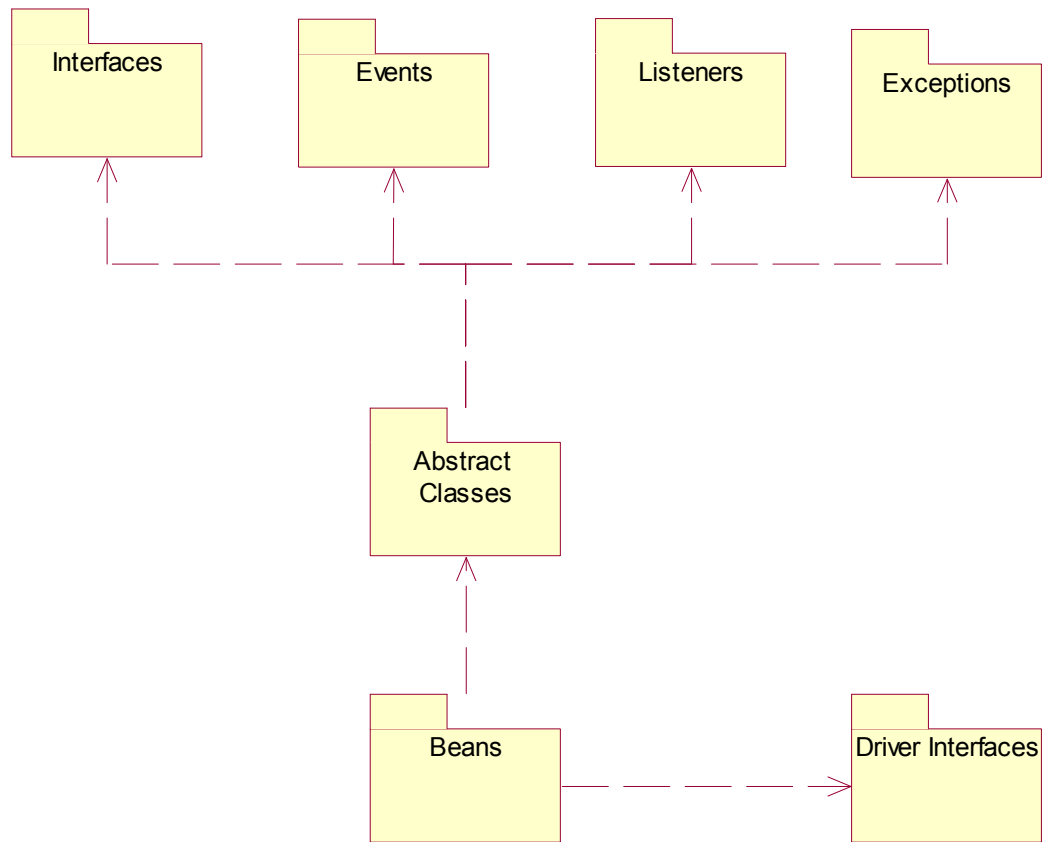


## A.2. Framework Design (Class Package Diagram)

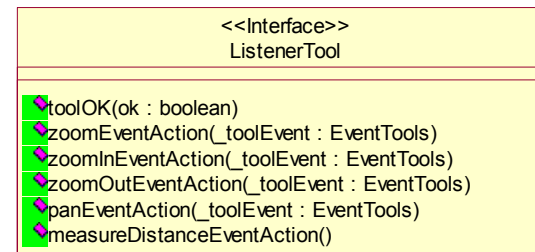
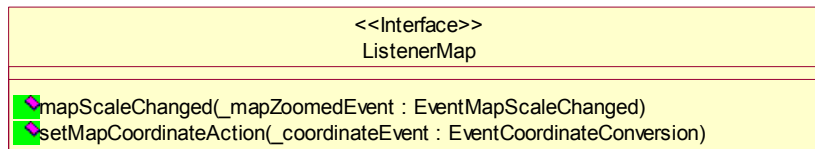
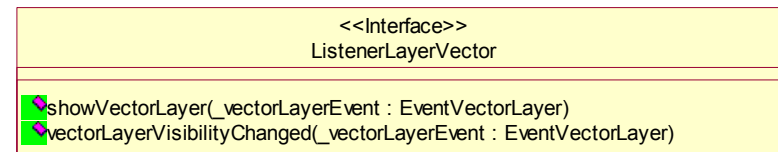
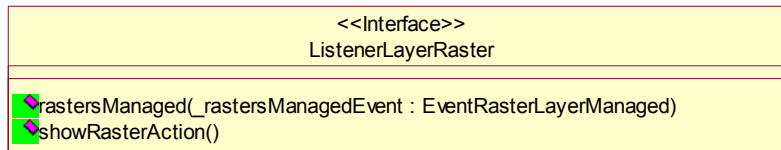
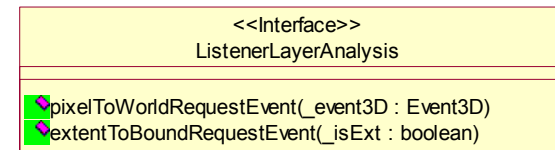
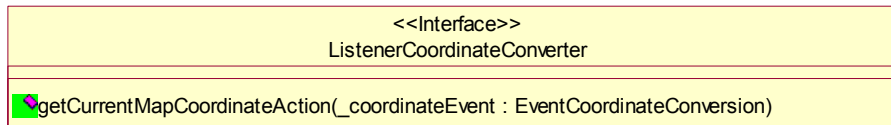




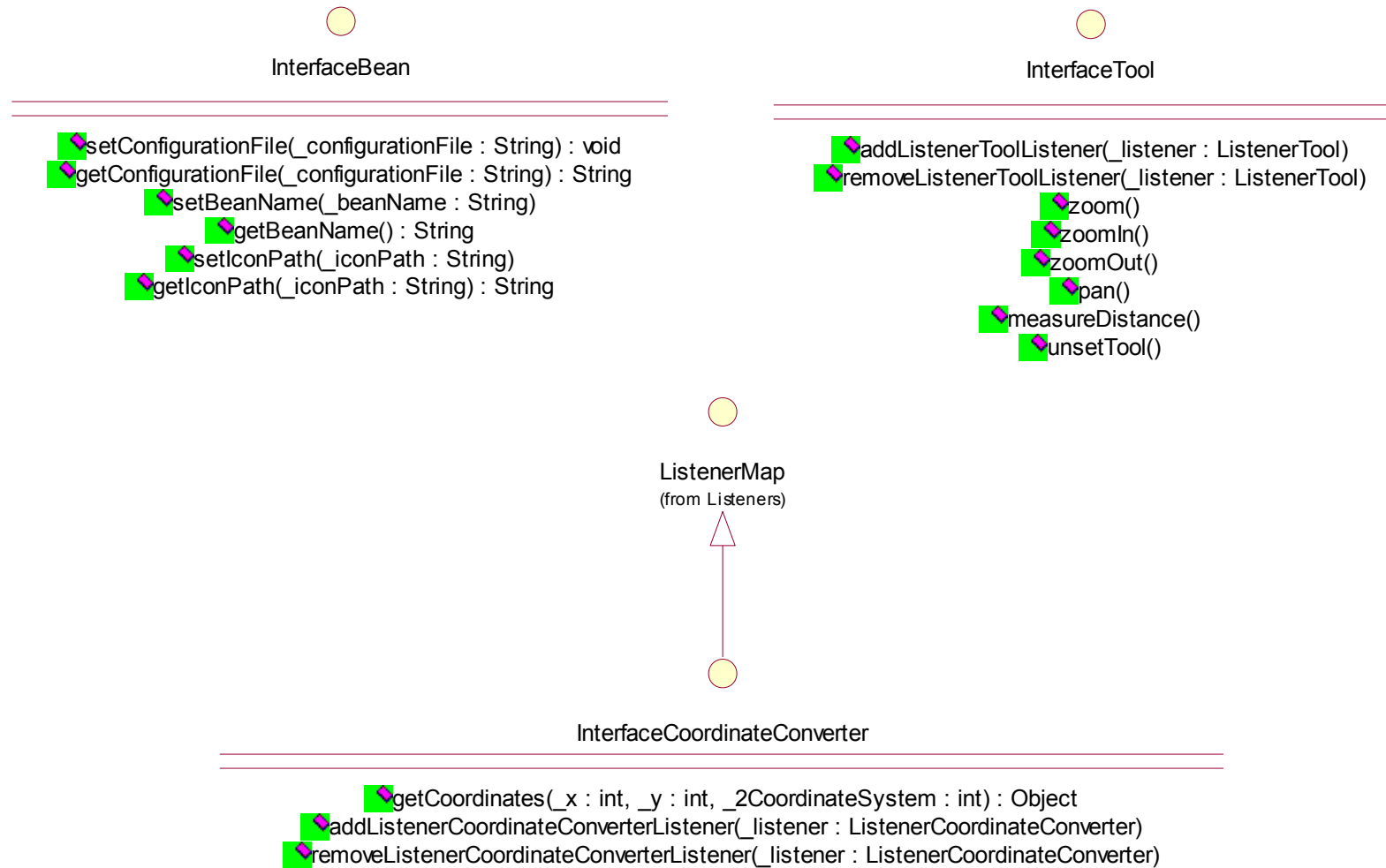
### A.3. Basic Data Structures in the Framework

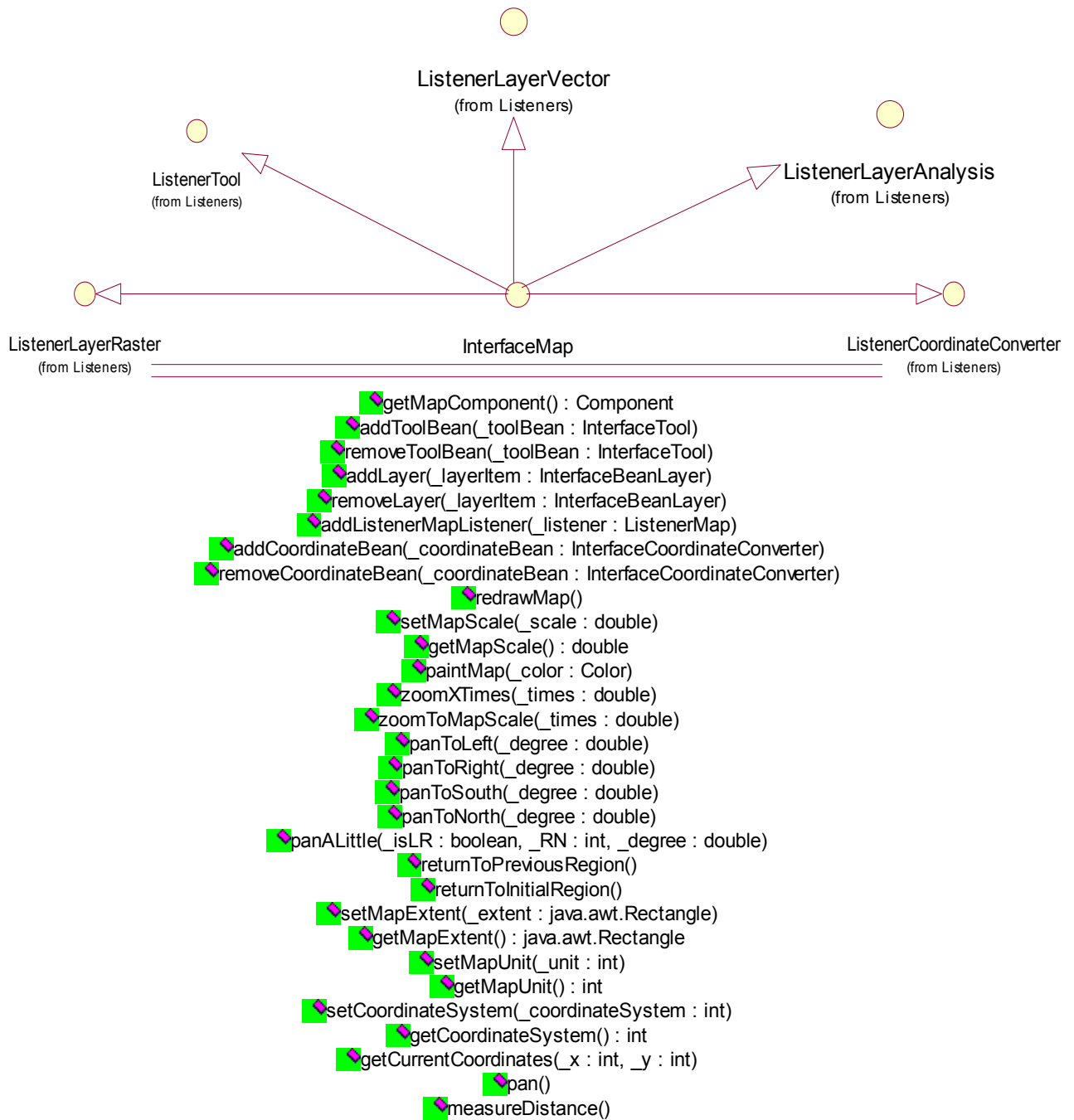


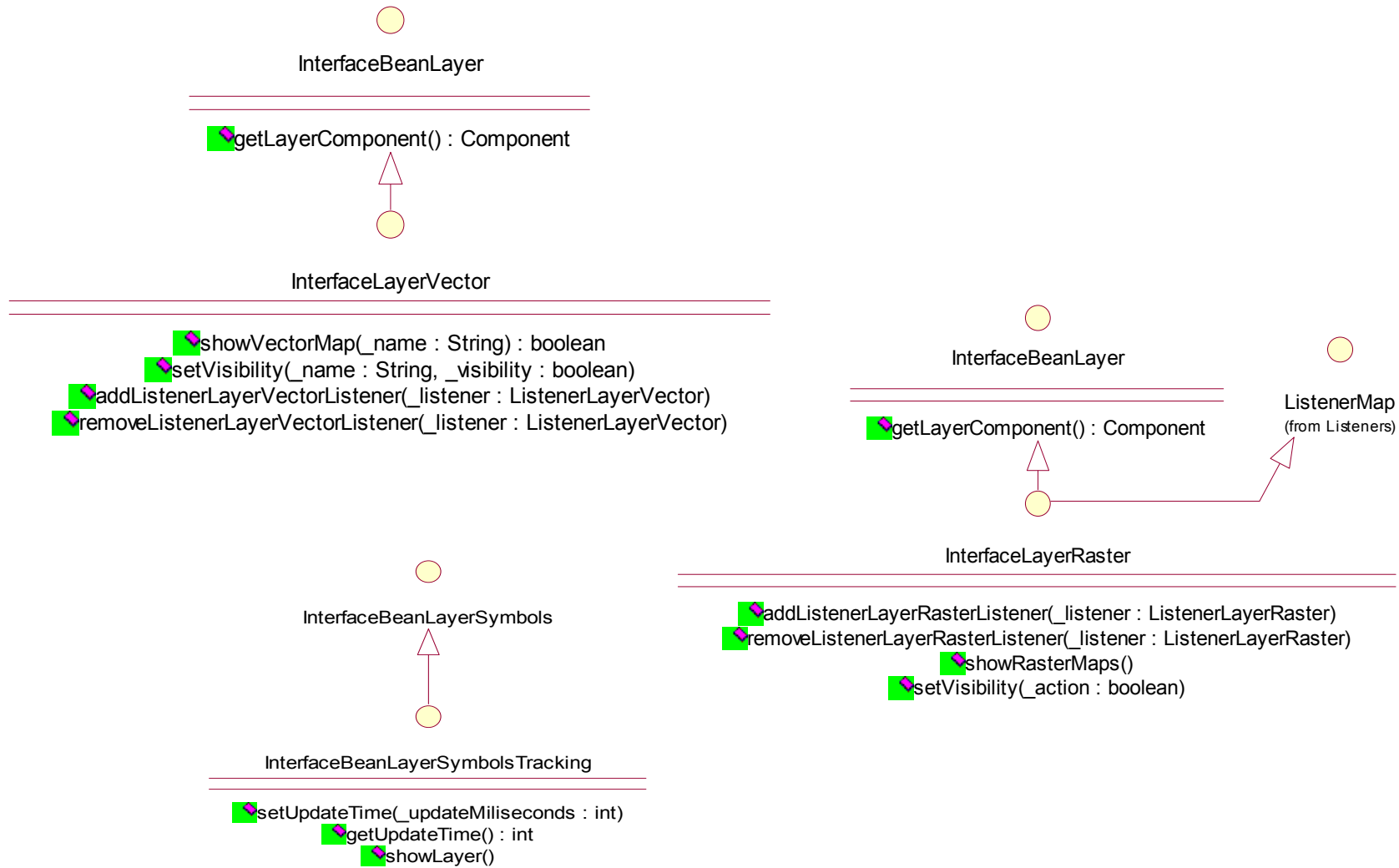
## Listeners

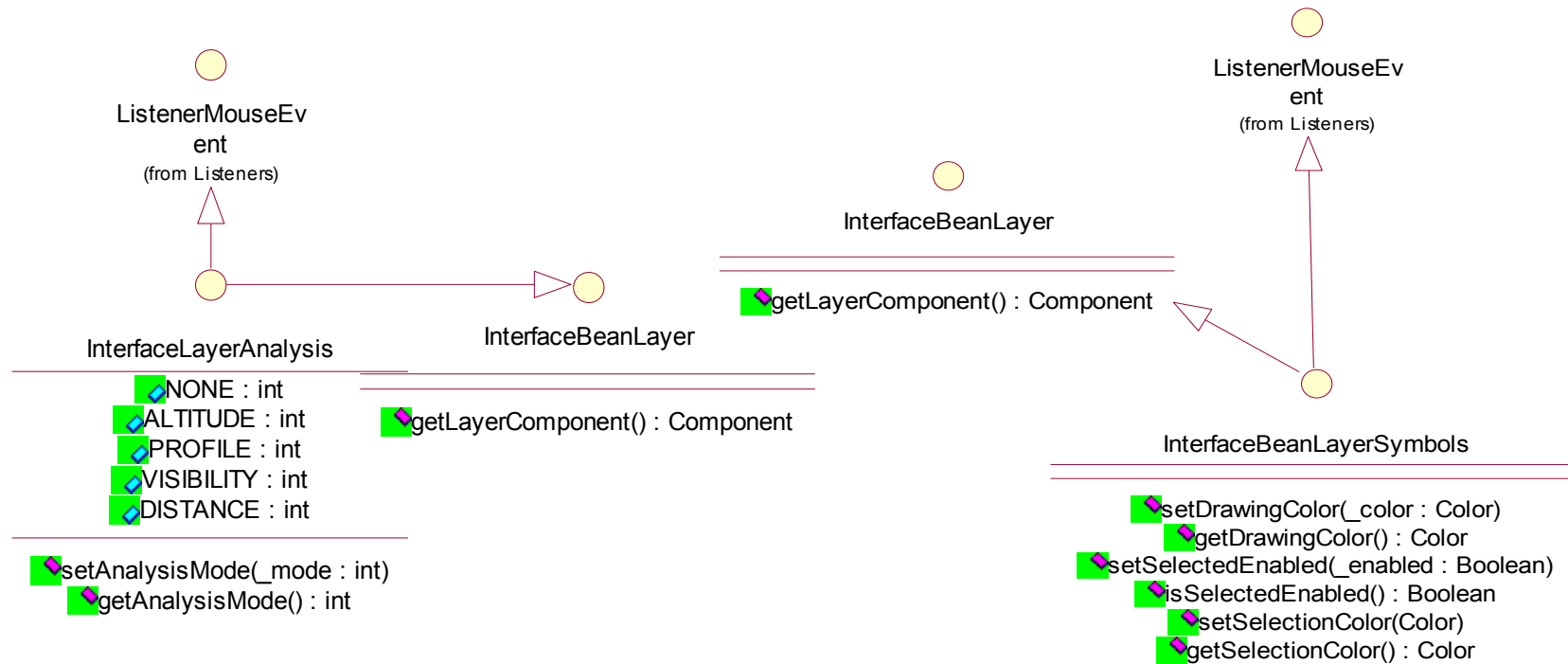


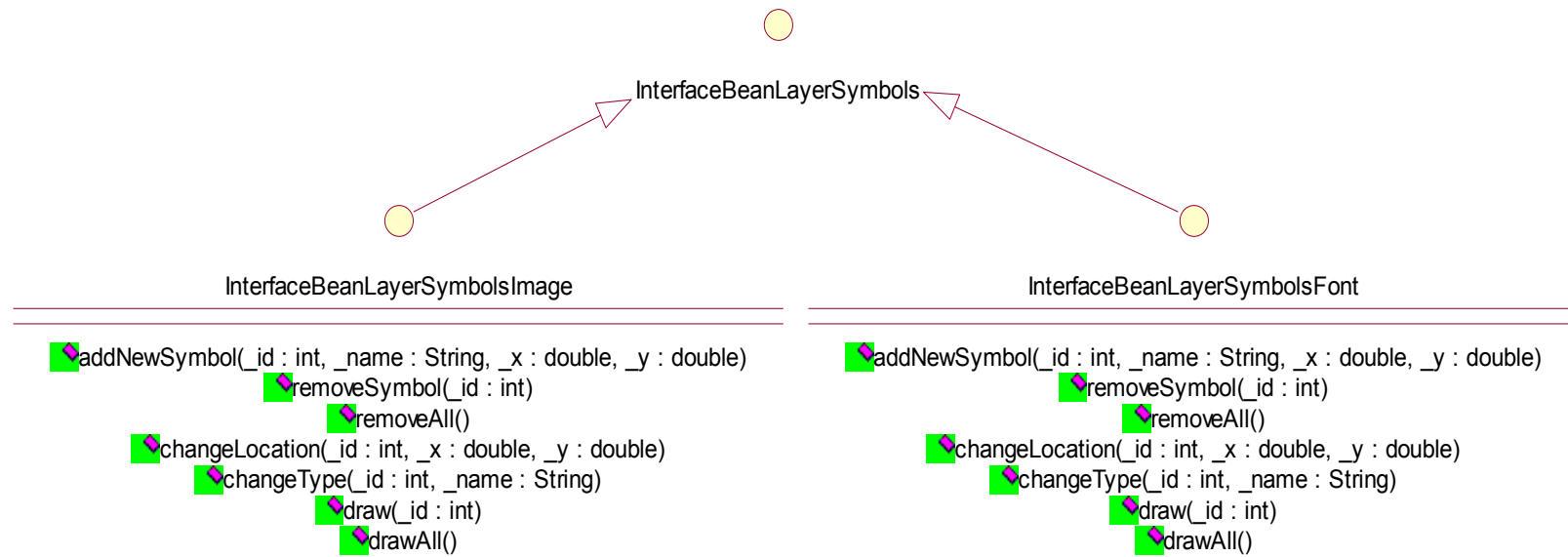
## Interfaces

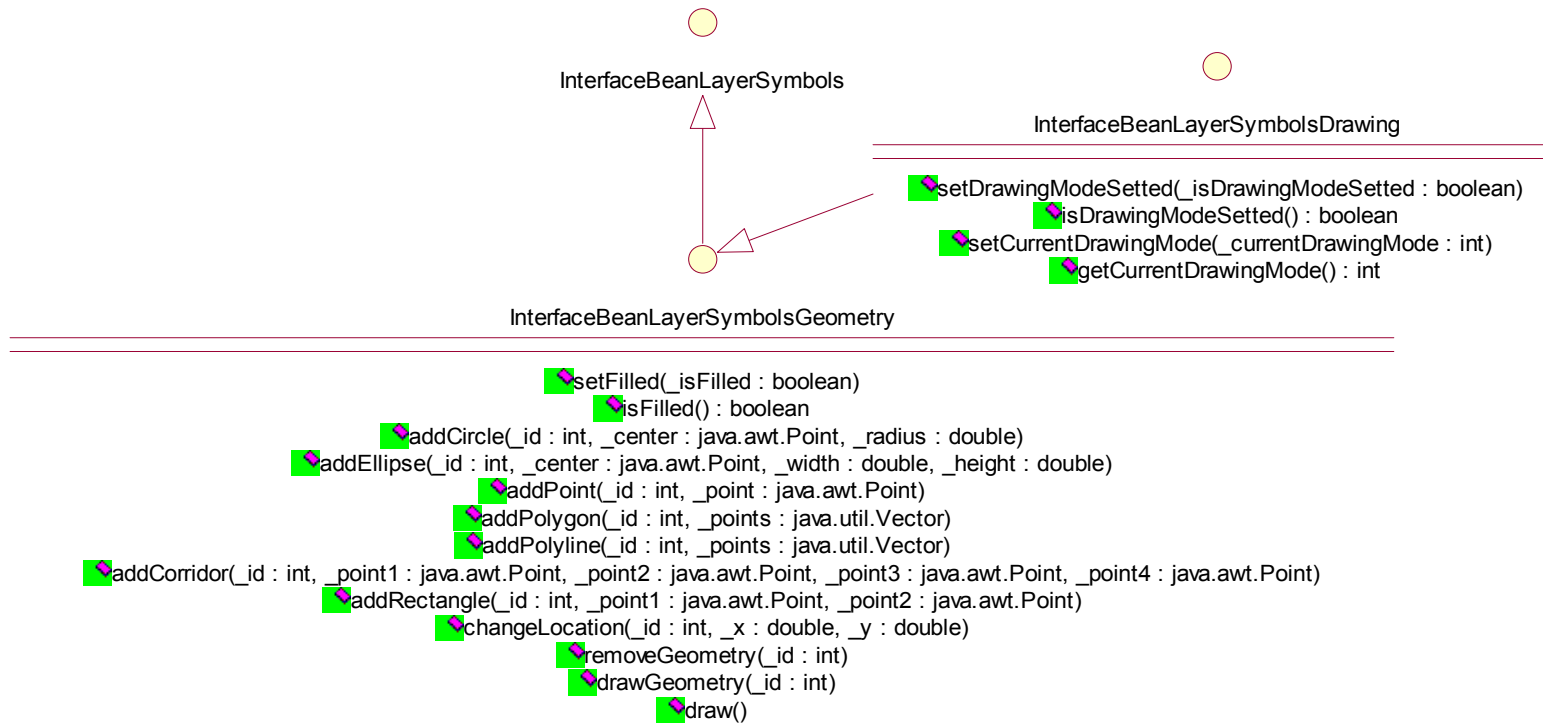






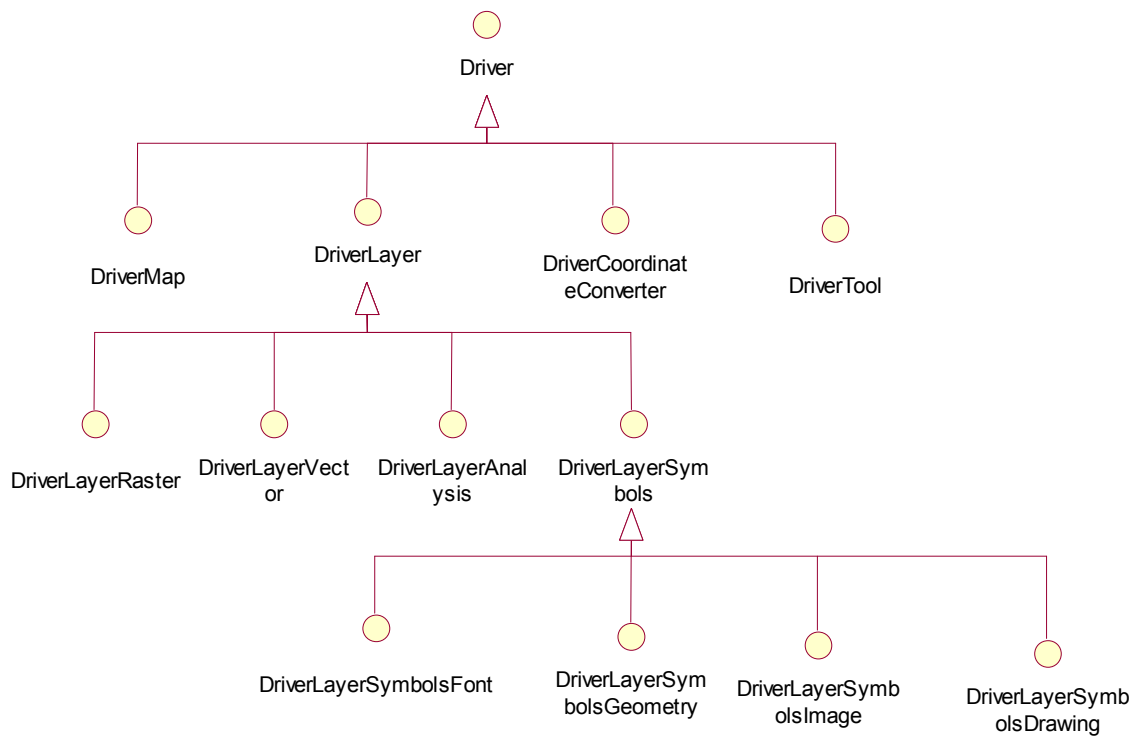




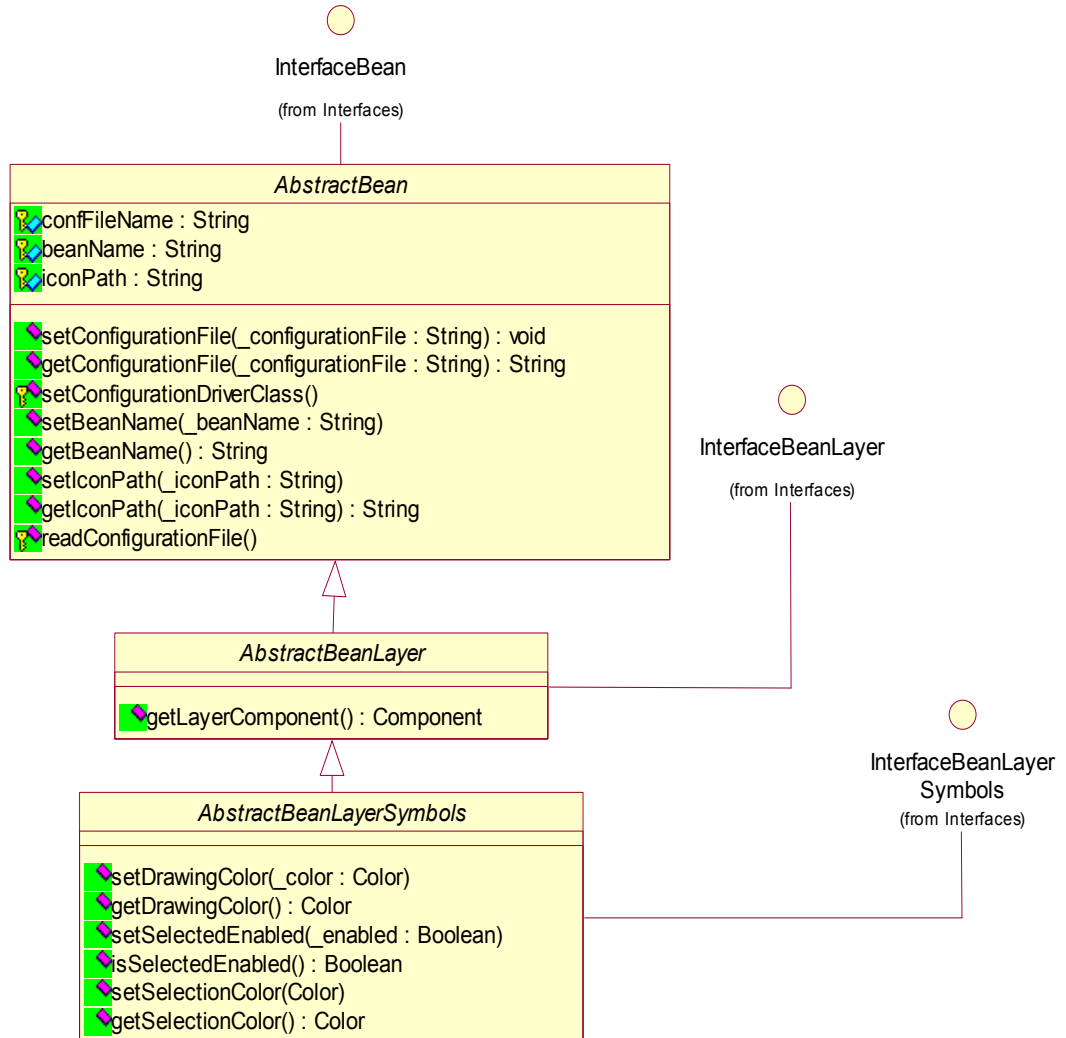




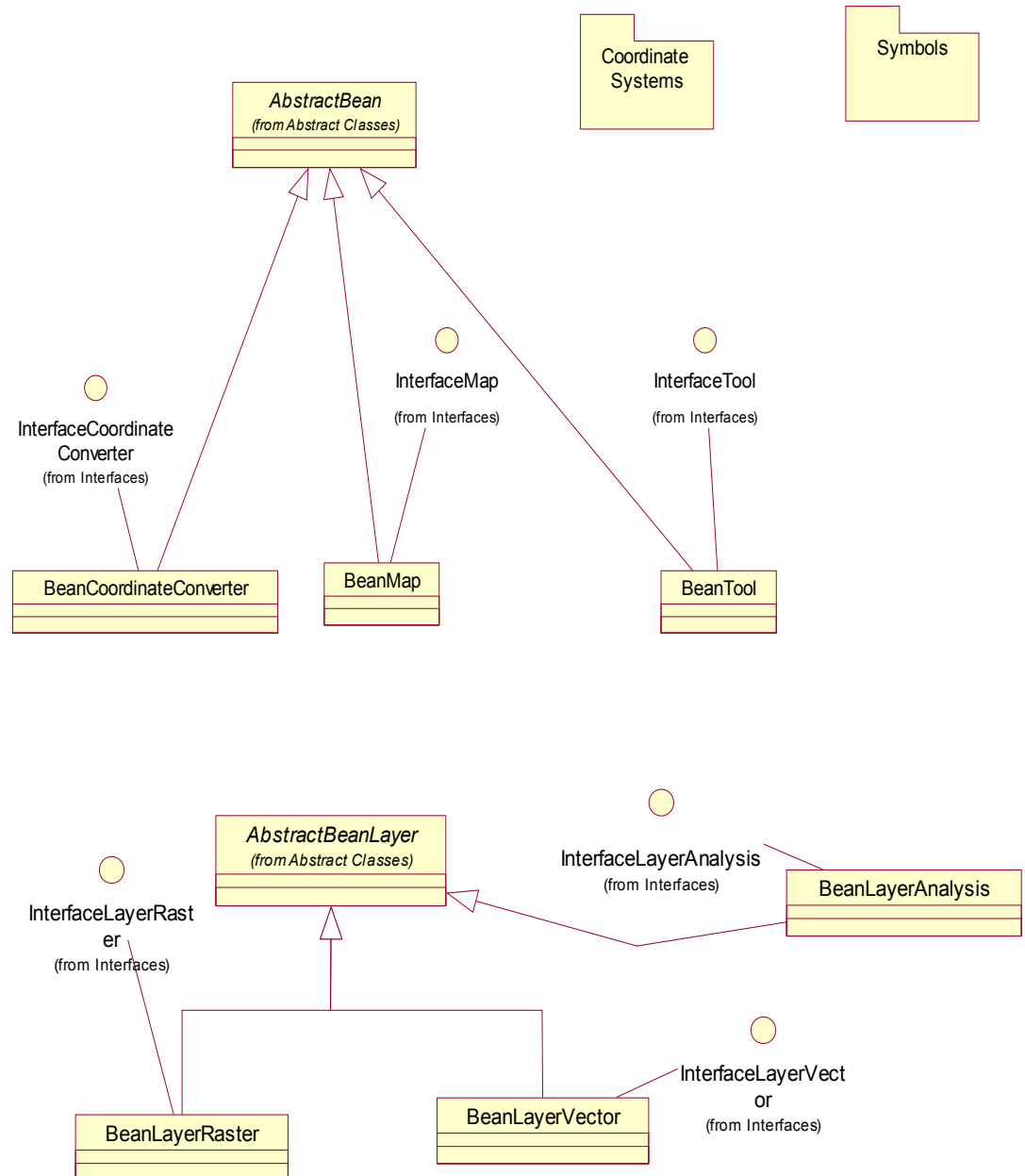
## Driver Interfaces

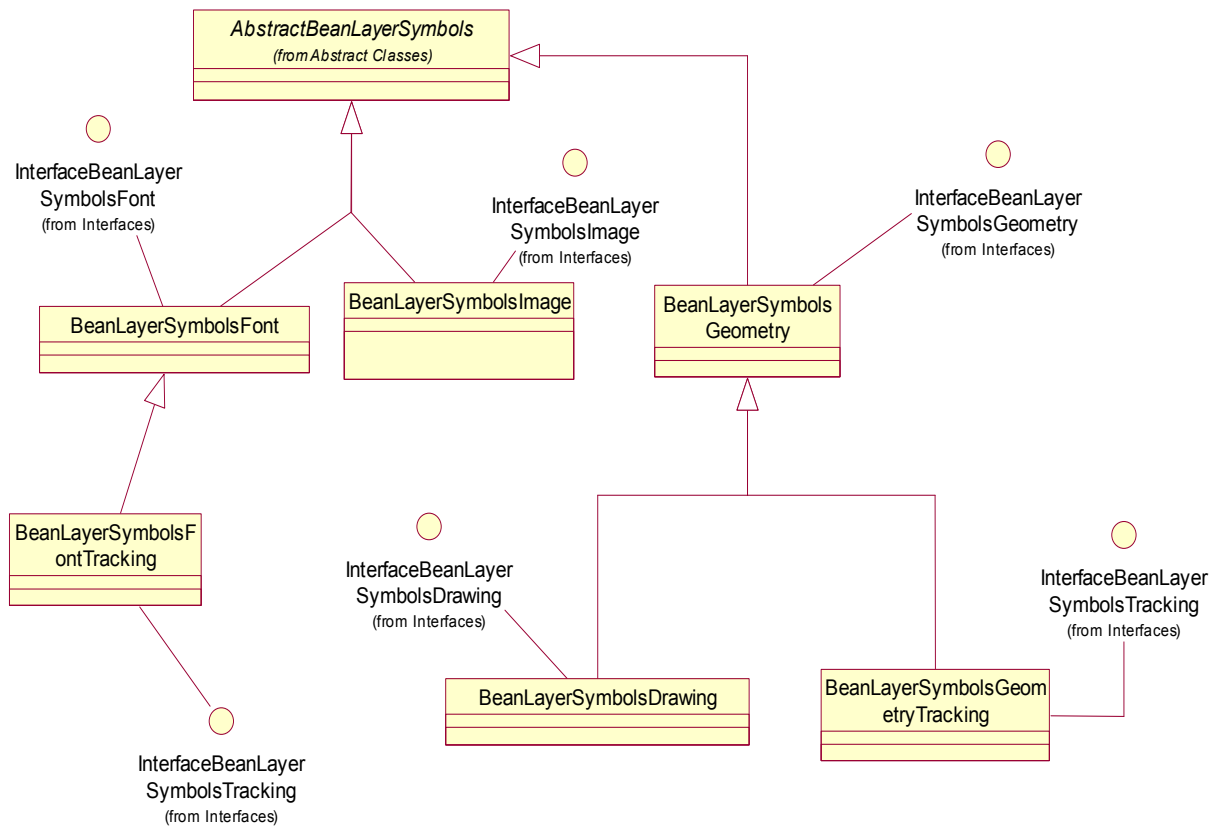


## Abstract Classes



## Beans

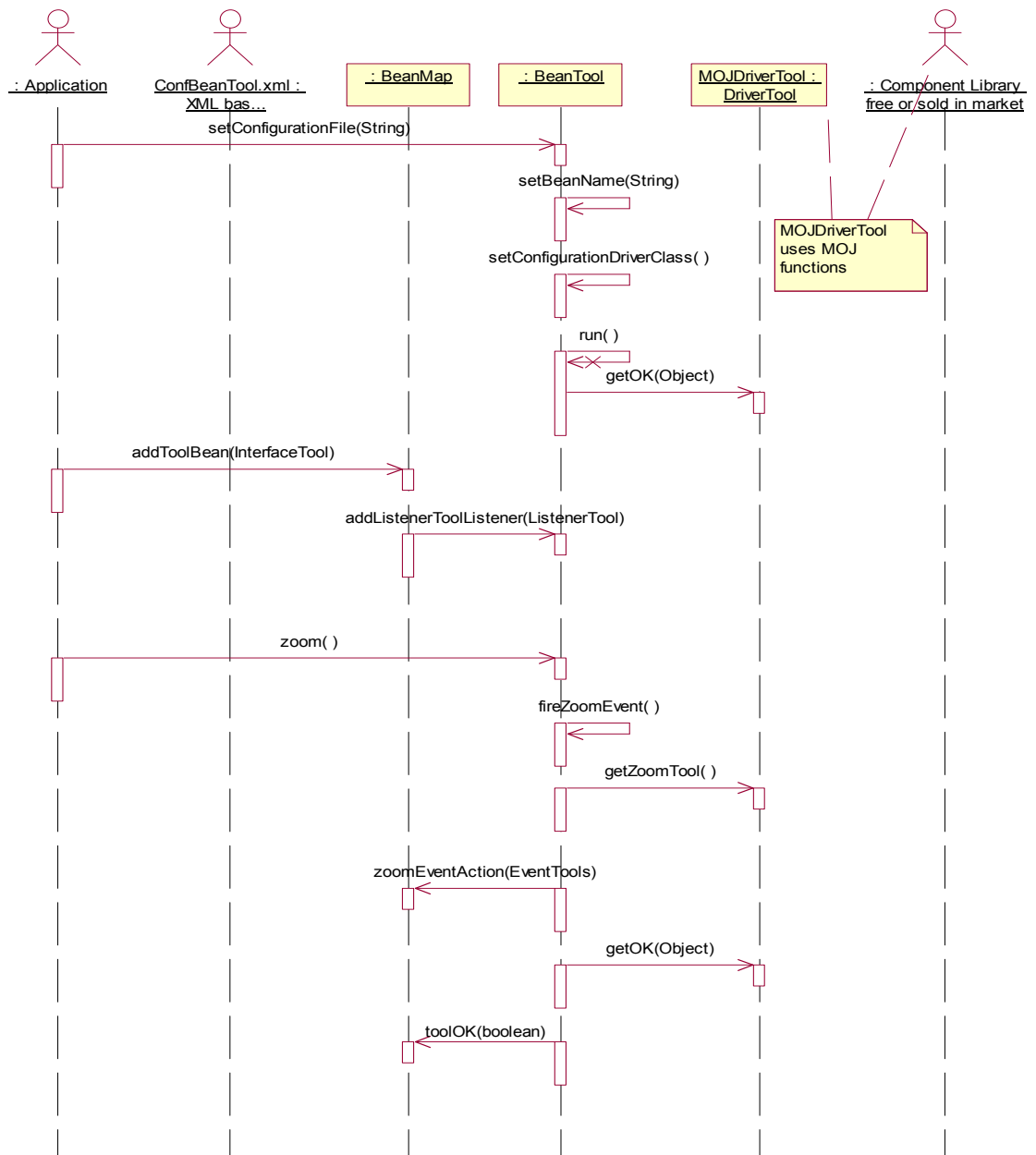




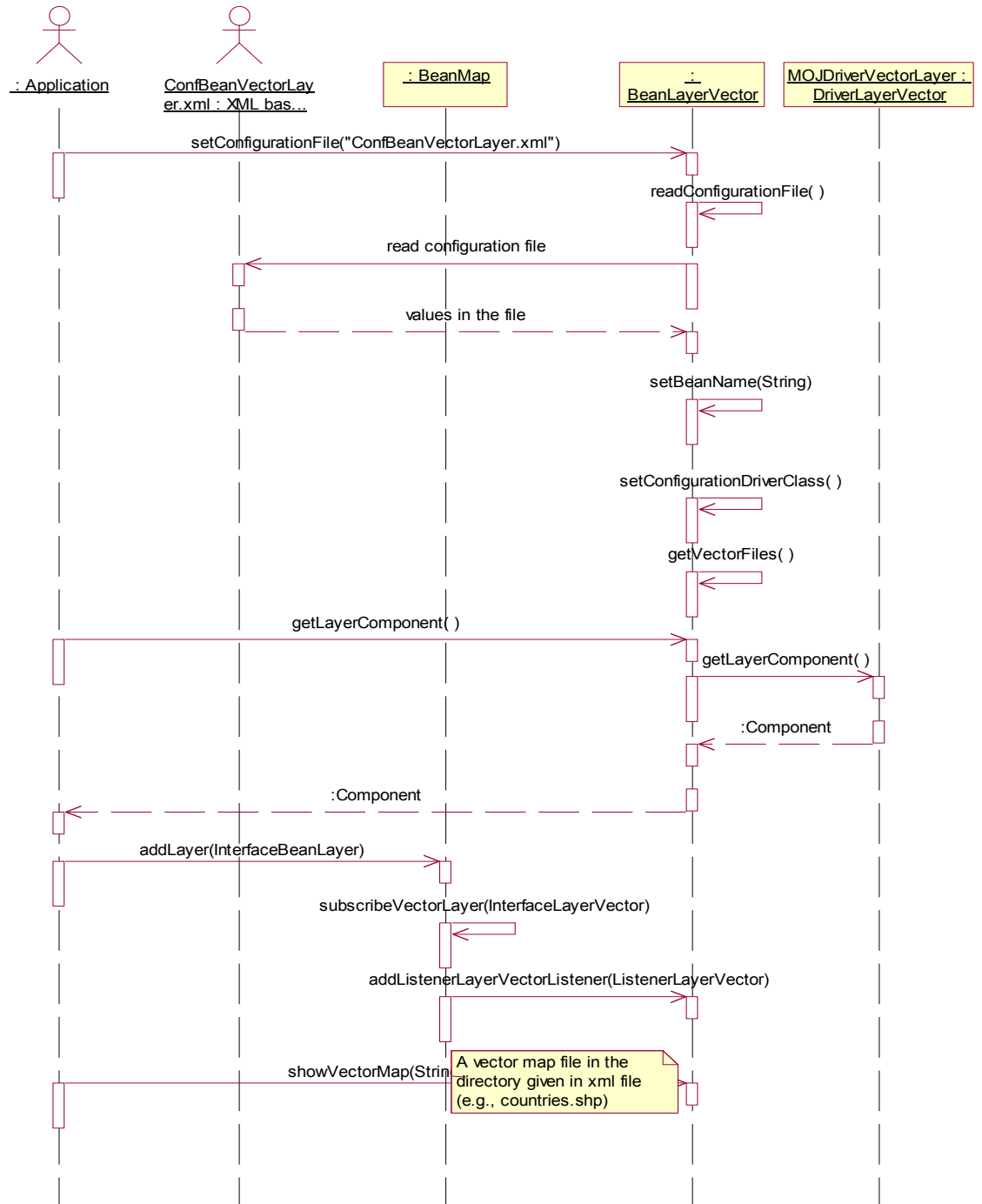
## B.4. Sequence Diagrams of Interactions between Components

This section includes sequence diagrams that show interactions between the developed beans for the GIS Domain Framework.

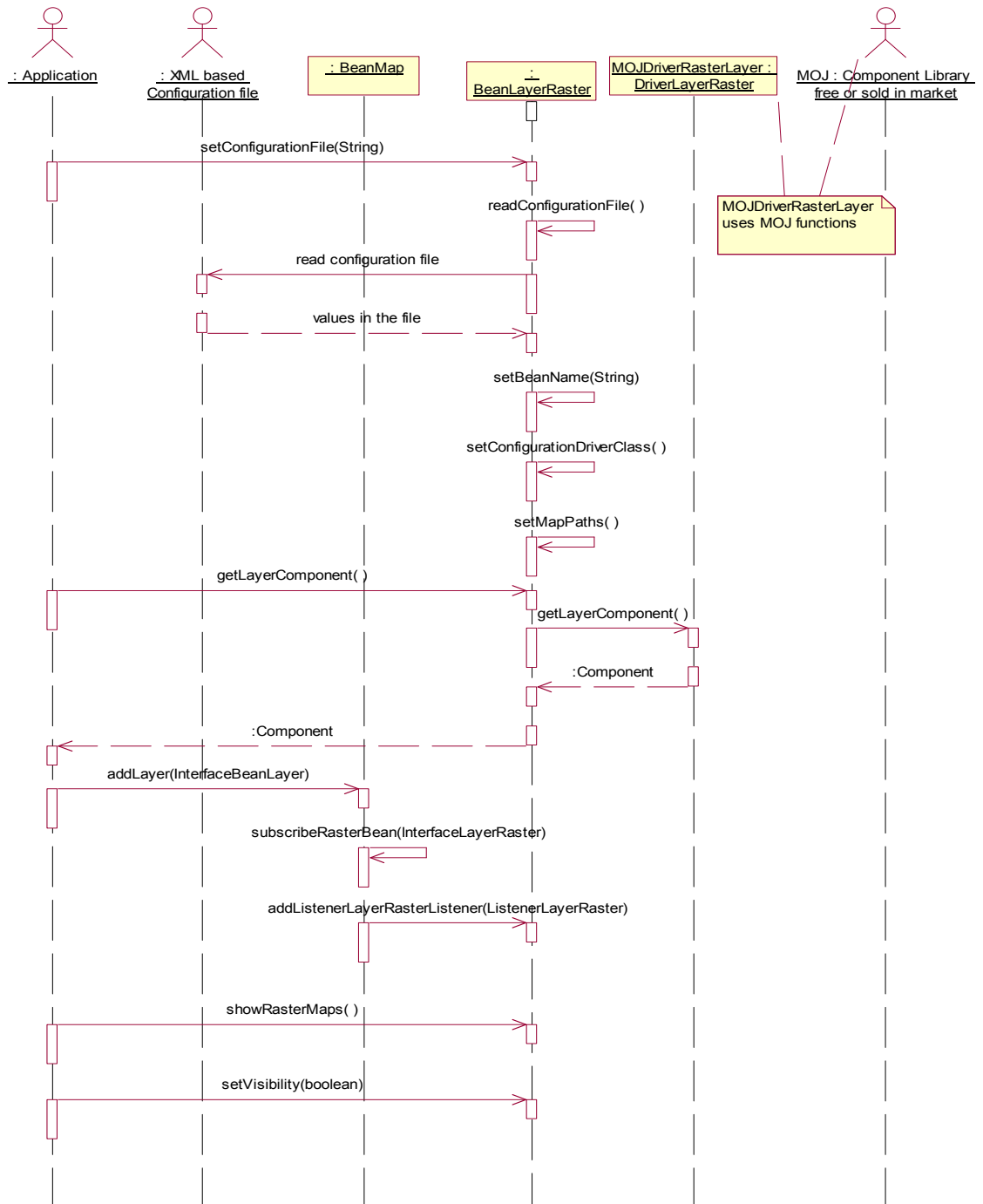
### Interaction between BeanMap and BeanTool Components



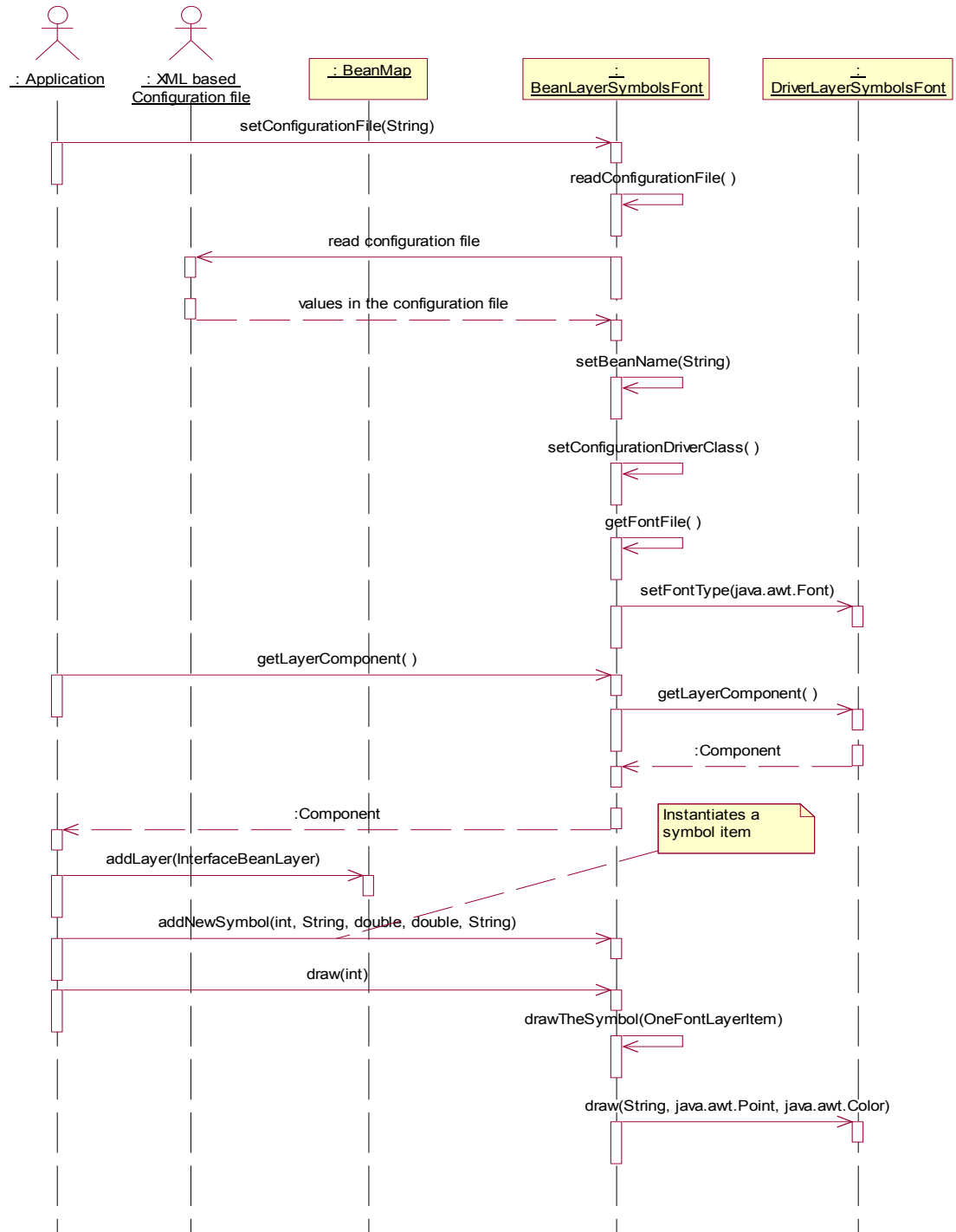
## Interaction between BeanMap and BeanLayerVector Components



## Interaction between BeanMap and BeanLayerRaster Components

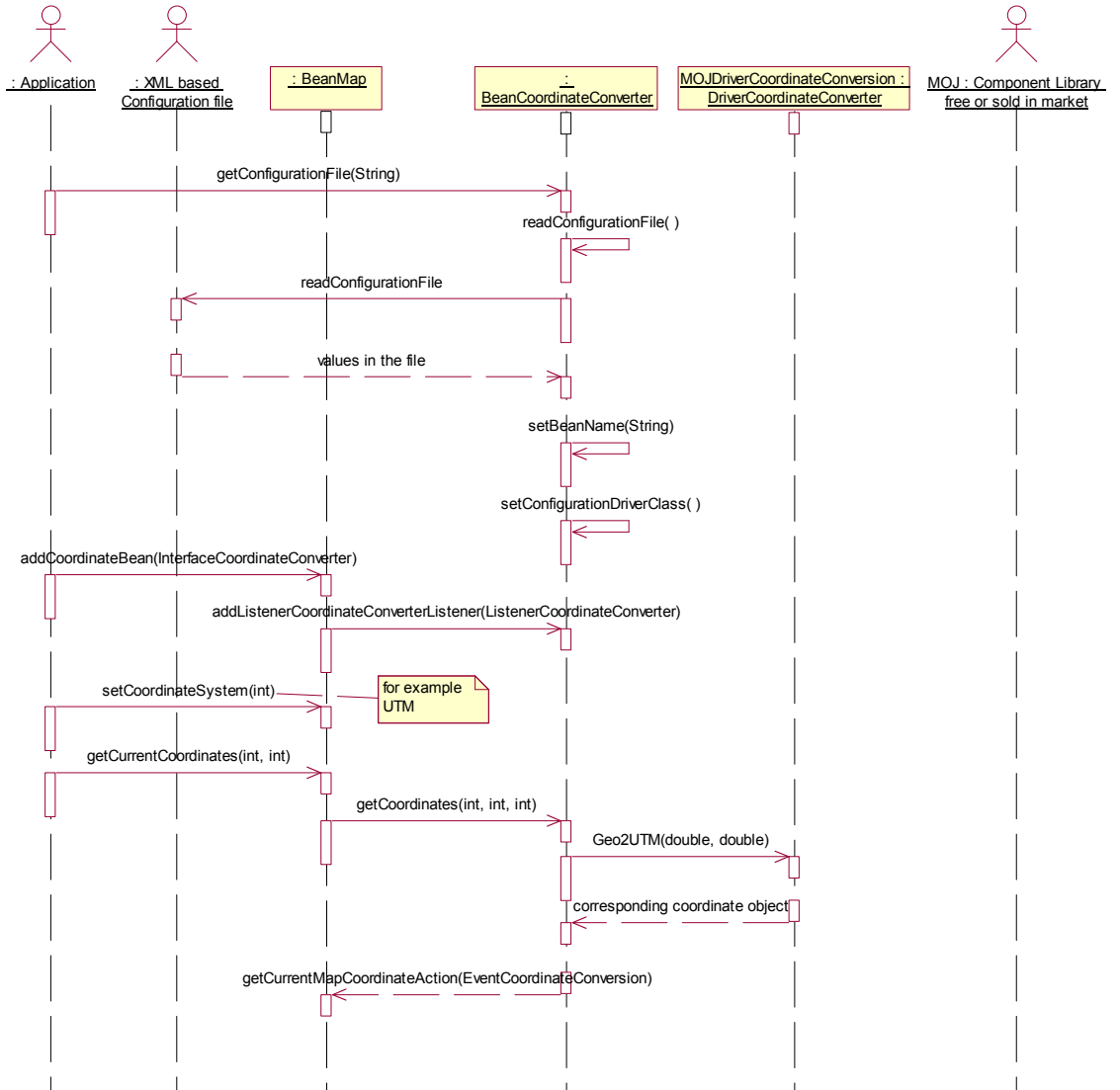


## Interaction between BeanMap and BeanLayerSymbolsFont Components

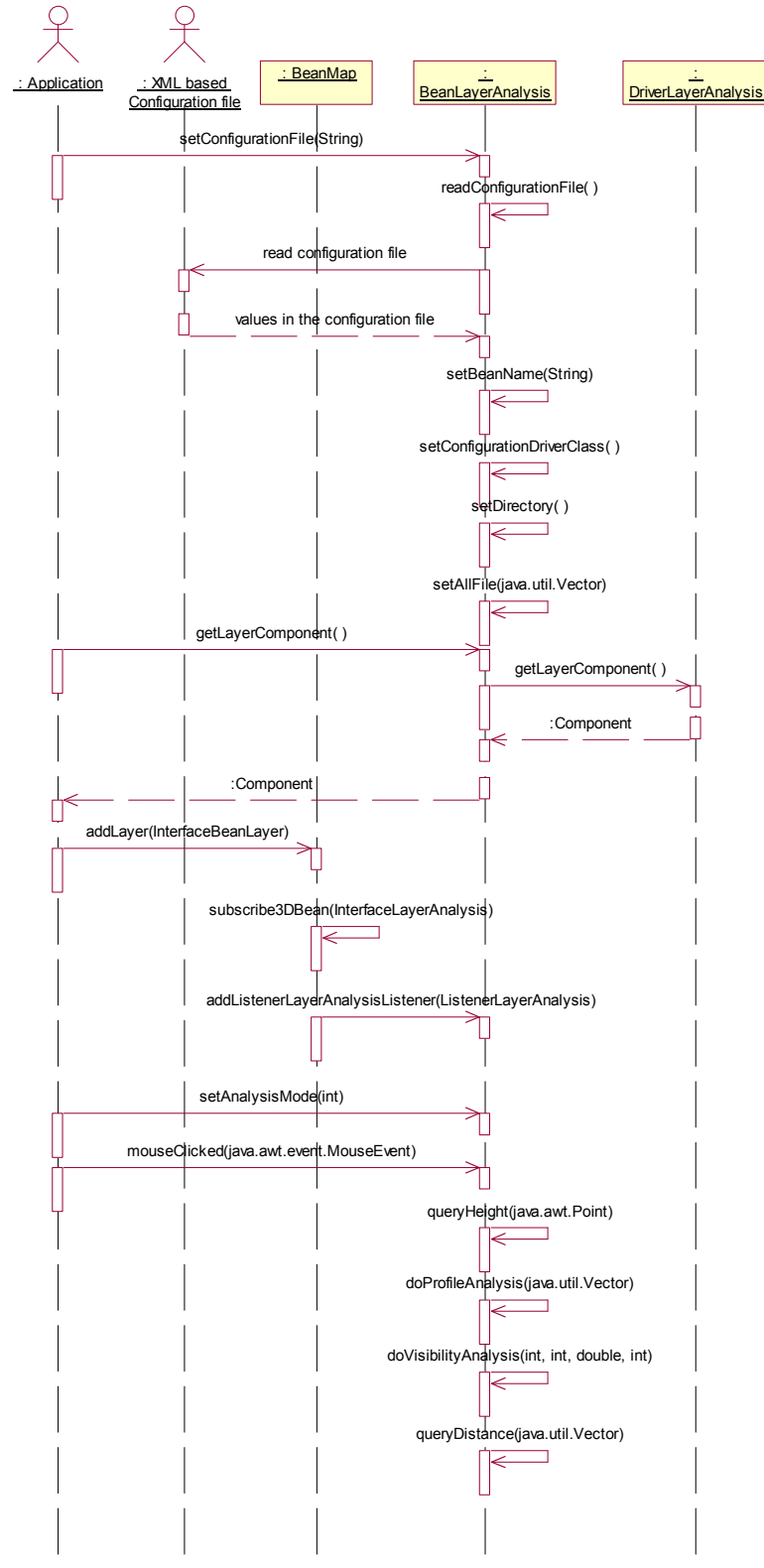




## Interaction between BeanMap and BeanCoordinateConverter Components



## Interaction between BeanMap and BeanLayerAnalysis Components



## APPENDIX B

### DETAILED DEFINITION OF DEVELOPED BEAN CLASSES IN GIS DOMAIN FRAMEWORK

#### B.1. Interfaces for Abstract Classes

##### InterfaceBean

*Description:* This is an interface that describes methods, which a bean in this framework must implement

*Methods:*

setConfigurationFile(String \_configurationFile) throws ExceptionBean: A bean must implement this function to set configuration file which a bean must read.

String getConfigurationFile() throws ExceptionBean: A bean must implement this function to get configuration file name.

setBeanName(String \_beanName) throws ExceptionBean: A bean must implement this function to set bean name.

String getBeanName() throws ExceptionBean: A bean must implement this function to get bean name.

setIconPath(String \_iconPath) throws ExceptionBean: A bean must implement this function to set bean icon path.

String getIconPath() throws ExceptionBean: A bean must implement this function to get bean icon path.

## **InterfaceBeanLayer**

**Description:** This is an interface that must be implemented by a layer bean in this framework. It offers a function, `getLayerComponent`, which a layer bean in this framework must have.

### **Methods:**

`java.awt.Component getLayerComponent()` throws `ExceptionBeanLayer`: A layer bean must implement this function to get layer component.

## **InterfaceBeanLayerSymbols**

**Description:** This is an interface that must be implemented by a symbol layer bean in this framework. Because of being a layer, which is used to show symbols, this interface is extended from `InterfaceBeanLayer`. This interface also extended from `ListenerMouseEvent` listener to be notified by mouse events.

### **Methods:**

`setColor(java.awt.Color _color)` throws `ExceptionBeanLayerSymbols`: A symbols layer bean must implement this function to set symbol drawing color.

`java.awt.Color getColor()`: A symbols layer bean must implement this function to get symbol drawing color.

`setSelectedEnabled (boolean _enabled)`: A symbols layer bean must implement this function to set mouse selection property enabled.

`boolean getSelectedEnabled ()`: A symbols layer bean must implement this function to get mouse selection property.

`setSelectionColor(java.awt.Color _color)`: A symbols layer bean must implement this function to set symbol selection color.

`java.awt.Color getSelectionColor()`: A symbols layer bean must implement this function to get symbol selection color.

## **InterfaceBeanLayerSymbolsTracking**

**Description:** This is an interface that must be implemented by a tracking symbol layer bean in this framework. Because of being a layer, which is used to show symbols, this interface is extended from InterfaceBeanLayerSymbols. This interface also extended from Runnable interface to move objects.

### **Methods:**

setUpdateTime(int \_updateMiliseconds) throws ExceptionBeanLayer: This method is used to set the refresh time for the objects on the layer.

getUpdateTime() throws ExceptionBeanLayer: This method is used to get the refresh time for the objects on the layer.

showLayer() throws ExceptionBeanLayer: this method is used to start moves of the objects on the layer.

## **B.2. Abstract Classes**

### **AbstractBean**

**Description:** This class describes general bean class. It implements InterfaceBean interface. All beans in this framework are children of AbstractBean class. This class implements all generic functions for the beans. A bean includes a Driver object. This object means an instance of driver class implementing Driver interface by using functions of a component library. This class read an XML based configuration file that includes specific properties for the bean. Application gives this file to AbstractBean class by using the function of this class.

### **AbstractBeanLayer**

**Description:** This class describes general layer bean class. It implements InterfaceBeanLayer interface and extended from AbstractBean, which is core class for all beans in this framework. All layer beans in this framework are children of AbstractBeanLayer class. This class implements all generic functions for the layer beans. A layer bean includes a DriverLayer object. This object means an instance of driver class implementing DriverLayer interface by using functions of a component library.

## **AbstractBeanLayerSymbols**

**Description:** This class describes general layer bean on which symbols are shown. It implements InterfaceBeanLayerSymbols interface and extended from AbstractBeanLayerSymbols, which is core class for layer beans in this framework. All symbol layer beans in this framework are children of AbstractBeanLayerSymbols class. This class implements all generic functions for the symbol layer beans.

## **B.3. Listeners**

### **ListenerMap**

**Description:** Beans that want to know events happened in Map bean implement this listener.

#### **Methods:**

mapScaleChanged: This event is used to notify changes in scale of the Map bean. If a zoom in/out action happens, the scale of the map changes.

setMapCoordinateAction: This event is used to notify listeners about setting map coordinate action is happened.

### **ListenerTool**

**Description:** Beans that want to know about events occurred in Tool Bean implement this listener.

#### **Methods:**

toolOK: Used to notify listeners about finishing tool action.

zoomEventAction: Used to notify listeners about zoom event action is occurred.

zoomInEventAction : Used to notify listeners about zoom in event action is occurred.

zoomOutEventAction : Used to notify listeners about zoom out event action is occurred.

panEventAction : Used to notify listeners about pan event action is occurred.

measureDistanceEventAction: Used to notify listeners about pan event action is occurred.

### **ListenerLayerVector**

***Description:*** Beans that want to know about events occurred in Vector Layer Bean implement this interface.

***Methods:***

showVectorLayer: Used to notify listeners about showing vector layer.

vectorLayerVisibilityChanged: Used to notify listeners about changing visibility of vector layer.

### **ListenerRasterLayer**

***Description:*** Beans that want to know about events occurred in Raster Layer Bean implement this interface.

***Methods:***

rastersManaged: Used to notify listeners about determination of showing raster layer directory according to map scale.

showRasterAction: Used to notify listeners about showing raster layer.

### **ListenerLayerAnalysis**

***Description:*** This listener is used to listen events occurred in analysis layer.

***Methods:***

pixelToWorldRequestEvent: Used to notify listeners about converting pixel coordinates to world coordinates event.

### **ListenerCoordinateConverter**

***Description:*** This listener is implemented by beans that want to know about events occurred in Coordinate converter bean.

***Methods:***

getCurrentMapCoordinateAction: Used to notify listeners about event finishing conversion of current map coordinates

## B.4. Interfaces

### InterfaceMap

**Description:** This is an interface class for BeanMap. Beans written for Map component in this framework must implement this interface. A Map bean works with Tool, Layer (Raster layer, Vector layer, Analysis layer, Symbols layer), and Coordinate Conversion beans. Therefore, InterfaceMap presents functions to add these beans. In addition a map bean listens some of these beans to make some actions according to the results of events occurred in these beans. Map bean must listen to these beans. Therefore, InterfaceMap extends the listeners of these layers.

A Map bean is the core bean in the framework. To use other beans in the framework, firstly these beans must be added to map bean. Therefore, InterfaceMap declares addToolBean, removeToolBean, addLayer, removeLayer, addCoordinateBean, and removeCoordinateBean functions.

#### **Methods:**

getMapComponent: A Map bean must implement getMapComponent() to get map component of used component library.

addListenerMapListener(ListenerMap \_listener): A Map bean must implement this function to add beans that want to listen events occurred in map bean.

removeListenerMapListener(ListenerMap \_listener): A Map bean must implement this function to remove beans that want to listen events occurred in map bean.

addToolBean(InterfaceTool \_toolBean): A Map bean must implement this function to add Tool bean that implements InterfaceTool interface.

removeToolBean(InterfaceTool \_toolBean): A Map bean must implement this function to remove Tool bean that implements InterfaceTool interface..

addLayer(InterfaceBeanLayer \_layerItem): A Map bean must implement this function to add any Layer bean that implements InterfaceBeanLayer interface.



removeLayer(InterfaceBeanLayer \_layerItem): A Map bean must implement this function to remove any Layer bean that implements InterfaceBeanLayer interface.

addCoordinateBean(InterfaceCoordinateConverter \_coordinateBean): A Map bean must implement this function to add any Coordinate Converter bean that implements InterfaceCoordinateConverter interface.

removeCoordinateBean(InterfaceCoordinateConverter \_coordinateBean): A Map bean must implement this function to remove any Coordinate Converter bean that implements InterfaceCoordinateConverter interface.

redrawMap: A Map bean must implement this function to redraw map bean and all layers registered to map bean.

setMapScale: A Map bean must implement this function to set scale of map bean.

getMapScale: A Map bean must implement this function to get scale of map bean.

paintMap: A Map bean must implement this function to paint map bean with the specified color.

zoomXTimes: A Map bean must implement this function to zoom map by specified times.

zoomToMapScale: A Map bean must implement this function to zoom map to specified scale.

panToLeft: A Map bean must implement this function to pan map to left by specified degree.

panToRight: A Map bean must implement this function to pan map to right by specified degree.

panToSouth: A Map bean must implement this function to pan map to south by specified degree.

panToNorth: A Map bean must implement this function to pan map to north by specified degree.

returnToPreviousRegion: A Map bean must implement this function to return map extent to previous.

setMapExtent: A Map bean must implement this function to set map extent

getMapExtent: A Map bean must implement this function to get map extent

returnToInitialRegion: A Map bean must implement this function to set initial map extent.

setMapUnit: A Map bean must implement this function to set map unit.

getMapUnit: A Map bean must implement this function to get map unit.

setCoordinateSystem(int \_coordinateSystem) throws ExceptionMap: A Map bean must implement this function to set coordinate system of map.

int getCoordinateSystem():A Map bean must implement this function to get coordinate system of map.

Object getCurrentCoordinates(int \_x, int \_y) throws ExceptionMap: A Map bean must implement this function to get world coordinates of pixel coordinate in coordinate system of map.

## **InterfaceTool**

**Description:** This is an interface that must be implemented by a tool bean in this framework. It offers functions, which a tool bean in this framework must have. This interface presents addListenerToolListener and removeListenerToolListener functions, which a tool bean must implement to add and remove Listeners that implements ListenerTool correspondingly.

### **Methods:**

addListenerToolListener(ListenerTool \_listener): A Tool bean must implement this function to add beans that want to listen events occurred in tool bean.

removeListenerToolListener(ListenerTool \_listener): A Tool bean must implement this function to remove beans that want to listen events occurred in tool bean.

zoom: A tool bean must implement this function to set zoom in/out tool and zoom in/out map while clicking the mouse.

zoomIn: A tool bean must implement this function to set zoom in tool and zoom in map.

zoomOut: A tool bean must implement this function to set zoom out tool and zoom out map.

pan: A tool bean must implement this function to set pan tool and pan map.

measureDistance: A tool bean must implement this function to set distance tool and measure distance between selected points selected by clicking mouse.

unsetTool: A tool bean must implement this function to unset selected tool.

## **InterfaceLayerVector**

**Description:** This is an interface that must be implemented by a vector layer bean in this framework must implement. Because of being used as interface of vector layer bean, this interface is extended from InterfaceBeanLayer interface. It offers functions, which a vector layer bean in this framework must have. A vector layer bean must implement addListenerLayerVectorListener and removeListenerLayerVectorListener methods to add and remove Listeners that implements ListenerLayerVector correspondingly.

### **Methods:**

addListenerLayerVectorListener(ListenerLayerVector \_listener): A Vector Layer bean must implement this function to add beans that want to listen events occurred in vector layer bean.

removeListenerLayerVectorListener(ListenerLayerVector \_listener): A Vector Layer bean must implement this function to remove beans that want to listen events occurred in vector layer bean.

showVectorMap(String \_name) throws ExceptionLayerVector: A Vector Layer bean must implement this function to show vector map named as “\_named” in the layer.

setVisibility(String \_name, boolean \_visibility) throws ExceptionLayerVector:  
A Vector Layer bean must implement this function to set visibility of vector map named as “\_named” in the layer to given “\_visibility” parameter.

### **InterfaceLayerRaster**

**Description:** This is an interface, which a raster layer bean in this framework must implement. Because of being used as interface of raster layer bean, this interface is extended from InterfaceBeanLayer. It offers functions, which a raster layer bean in this framework must have. A raster layer bean must implement addListenerLayerRasterListener and removeListenerLayerRasterListener functions to add and remove Listeners that implements ListenerLayerRaster correspondingly. A raster layer should listen map-based events to be notified when the scale of the map changes. Therefore, this interface is also extended from ListenerMap to be registered to map bean.

#### **Methods:**

addListenerLayerRasterListener(ListenerLayerRaster \_listener) throws ExceptionLayerRaster: A Raster Layer bean must implement this function to add beans that want to listen events occurred in raster layer bean.

removeListenerLayerRasterListener(ListenerLayerRaster \_listener) throws ExceptionLayerRaster: A Raster Layer bean must implement this function to remove beans that want to listen events occurred in raster layer bean.

showRaster() throws ExceptionLayerRaster: A Raster Layer bean must implement this function to show raster maps in the directory given in the XML based configuration file.

setVisibility(boolean \_action) throws ExceptionLayerRaster: A Raster Layer bean must implement this function to set visibility of raster maps in the directory given in the XML based configuration file given “\_visibility” parameter.

### **InterfaceLayerSymbolsGometry**

**Description:** This is an interface that must be implemented by a geometry layer bean in this framework. Because of being used as interface of geometry layer bean, which is a bean for showing symbols in the framework, this interface is extended from

InterfaceBeanLayerSymbols that is extended from InterfaceBeanLayer. This layer is also extended from ListenerMouseEvent to capture mouse-clicked events to select a geometry item. The interface offers functions, which a geometry layer bean in this framework must have.

***Methods:***

setIsFiiled(boolean \_isFilled) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to set filled geometric symbol type to draw filled geometric symbol.

getIsFiiled():A Geometry Layer bean must implement this function to get state of filled geometric symbol type.

addCircle(int \_id,java.awt.Point \_center,double \_radius)throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add circular geometric symbol to the layer.

addEllipse(int \_id, java.awt.Point \_center, double \_width, double \_height) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add elliptic geometric symbol to the layer.

addPoint(int \_id, java.awt.Point \_point) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add point symbol to the layer.

addPolygon(int \_id, java.util.Vector \_points) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add polygon symbol to the layer.

addPolyline(int \_id, java.util.Vector \_points) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add polyline symbol to the layer.

addCorridor(int \_id, java.awt.Point \_point1, java.awt.Point \_point2,java.awt.Point \_point3, java.awt.Point \_point4) throws ExceptionLayerSymbolsGeometry: A Geometry Layer bean must implement this function to add corridor symbol to the layer.

`addRectangle(int _id, java.awt.Point _point1, java.awt.Point _point2)` throws `ExceptionLayerSymbolsGeometry`: A Geometry Layer bean must implement this function to add rectangle symbol to the layer.

`changeLocation(int _id, double _x, double _y)` throws `ExceptionLayerSymbolsGeometry`: A Geometry Layer bean must implement this function to change location of a symbol identified with `_id` to given `_x`, `_y` coordinate.

`removeGeometry(int _id)` throws `ExceptionLayerSymbolsGeometry`: A Geometry Layer bean must implement this function to remove a symbol identified with `_id`.

`drawGeometry(int _id)` throws `ExceptionLayerSymbolsGeometry`: A Geometry Layer bean must implement this function to draw the symbol, which is added to the layer previously, identified with `_id`.

`drawAll()` throws `ExceptionLayerSymbolsGeometry`: A Geometry Layer bean must implement this function to draw all the symbols added to the layer previously.

## **InterfaceLayerSymbolsDrawing**

**Description:** This is an interface, which a drawing layer bean in this framework must implement. Because of being used as interface of drawing layer bean which is a bean for drawing and showing geometric symbols in the framework, this interface is extended from `InterfaceLayerSymbolsGeometry`.

```
final static int NONE = -1;
```

```
final static int POINT = 0;
```

### **Fields:**

```
final static int LINE = 1;
```

```
final static int POLYLINE = 2;
```

```
final static int CIRCLE = 3;
```

```
final static int ELLIPSE = 4;
```

```
final static int RECTANGLE = 5;
```

final static int POLYGON = 6;

**Methods:**

setIsDrawingModeSetted(boolean \_isDrawingModeSetted): A Drawing Layer bean must implement this function to set mode to drawing.

getIsDrawingModeSetted():A Drawing Layer bean must implement this function to get drawing mode.

clearShapes() throws ExceptionLayerSymbolsDrawing: A Drawing Layer bean must implement this function to clear all the drawn symbols in the layer.

setCurrentDrawingMode(int \_currentDrawingMode) throws ExceptionLayerSymbolsDrawing: A Drawing Layer bean must implement this function to set current drawing mode to one of the fields given above.

**InterfaceLayerSymbolsFont**

**Description:** This is an interface, which a font layer bean in this framework must implement. Because of being used as interface for font layer bean that is used to show symbols in the layer, this interface is extended from InterfaceBeanLayerSymbols. The interface offers functions, which a font layer bean in this framework must have.

**Methods:**

addNewSymbol(int \_id, String \_name, double \_x, double \_y) throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to add a symbol identified with “\_id” and “\_name” parameter which is defined with a font in the XML based configuration file.

removeSymbol(int \_id) throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to remove a symbol identified with “\_id” added to layer previously.

removeAll() throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to remove all the symbols added to layer previously.

changeLocation(int \_id, double \_x, double \_y) throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to change location of a symbol identified with \_id to given \_x, \_y coordinate.

draw(int \_id) throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to draw the symbol, which is added to the layer previously, identified with \_id.

drawAll() throws ExceptionLayerSymbolsFont: A Font Layer bean must implement this function to draw all the symbols added to the layer previously.

### **InterfaceLayerSymbolsImage**

**Description:** This is an interface that must be implemented by an image layer bean in this framework. Because of being used as interface for image layer bean, which is used to show symbols in the layer, this interface is extended from InterfaceBeanLayerSymbols. The interface offers functions, which an image layer bean in this framework must have.

#### **Methods:**

addNewSymbol(int \_id, String \_name, double \_x, double \_y) throws ExceptionLayerImage: An Image Layer bean must implement this function to add a symbol identified with “\_id” and “\_name” parameter which is defined with a font in the XML based configuration file.

removeSymbol(int \_id) throws ExceptionLayerImage: An Image Layer bean must implement this function to remove a symbol identified with “\_id” added to layer previously.

removeAll() throws ExceptionLayerImage: An Image Layer bean must implement this function to remove all the symbols added to layer previously.

changeLocation(int \_id, double \_x, double \_y) throws ExceptionLayerImage: An Image Layer bean must implement this function to change location of a symbol identified with \_id to given \_x, \_y coordinate.



`draw(int _id)` throws `ExceptionLayerImage`: An Image Layer bean must implement this function to draw the symbol, which is added to the layer previously, identified with `_id`.

`drawAll()` throws `ExceptionLayerImage`: An Image Layer bean must implement this function to draw all the symbols added to the layer previously.

### **InterfaceLayerAnalysis**

**Description:** This is an interface that must be implemented by an analysis layer bean in this framework. Because of being used as interface for analysis layer bean, this interface is extended from `InterfaceBeanLayer`. It offers functions, which an analysis layer bean in this framework must have. An analysis layer bean must implement `addListenerLayerAnalysisListener` and `removeListenerLayerAnalysisListener` methods to add and remove `ListenerLayerAnalysis` correspondingly. An analysis layer bean is extended from `ListenerMouseEvent` listener to be notified about mouse events.

#### **Fields:**

```
public final static int NONE = 0;

public final static int ALTITUDE = 1;

public final static int PROFILE = 2;

public final static int VISIBILITY = 3;

public final static int DISTANCE = 4;
```

#### **Methods:**

`addListenerLayerAnalysisListener(ListenerLayerAnalysis listener)` throws `ExceptionLayerAnalysis`: An Analysis Layer bean must implement this function to add beans that want to listen events occurred in analysis layer bean.

`removeListenerLayerAnalysisListener(ListenerLayerAnalysis listener)` throws `ExceptionLayerAnalysis`: An Analysis Layer bean must implement this function to remove beans that want to listen events occurred in analysis layer bean.

`setAnalysisMode(int _mode)` throws `ExceptionLayerAnalysis`: An Analysis Layer bean must implement this function to set analysis mode one of the fields above.

## **InterfaceCoordinateConverter**

**Description:** This is an interface, which coordinate converter bean in this framework must implement. A coordinate converter bean must implement `addListenerCoordinateConverterListener` and `removeListenerCoordinateConverterListener` methods to add and remove Listeners that implements `ListenerCoordinateConverter` correspondingly. A coordinate converter bean should listen to map based events. Therefore, this interface is also extended from `ListenerMap`. The interface offers functions, which a coordinate converter bean in this framework must have.

### **Fields:**

```
public static final int GEO  = 1;
```

```
public static final int UTM  = 2;
```

### **Methods:**

`addListenerCoordinateConverterListener(ListenerCoordinateConverter listener)` throws `ExceptionCoordinateConverter`: A Coordinate Converter bean must implement this method to add beans that want to listen events occurred in coordinate converter bean.

`removeListenerCoordinateConverterListener(ListenerCoordinateConverter listener)` throws `ExceptionCoordinateConverter`: A Coordinate Converter bean must implement this method to remove beans that want to listen events occurred in coordinate converter bean.

`Object getCoordinates(int _x, int _y, int _2CoordinateSystem)` throws `ExceptionCoordinateConverter`: A Coordinate Converter bean must implement this method to get coordinates of given pixel coordinates (`_x` and `_y`) in given `_2CoordinateSystem` coordinate system.

## **B.5. Driver Interfaces**

### **Driver**

**Description:** This is an interface which a driver class written to use component library in this framework implements.

## DriverMap

**Description:** This interface is used to supply requested functions by using component library function and implemented by a class. The map bean at runtime reads the implementing class and functions are invoked.

### **Fields:**

final static int NONE = -1;

final static int DEGREE = 0;

final static int METER = 1;

final static int FEET = 2;

### **Methods:**

java.awt.Component getMapComponent() throws ExceptionDriver: This function is implemented to get map component by using component library functions.

int addLayer(java.awt.Component \_layer) throws ExceptionDriver: This function is implemented to add layer component to map component by using component library functions.

removeLayer(java.awt.Component \_layer) throws ExceptionDriver: This function is implemented to remove layer component from map component by using component library functions.

setLayerVisibility(java.awt.Component \_layer, boolean \_visible) throws ExceptionDriver: This function is implemented to set visibility of layer component to \_visible by using component library functions.

java.awt.Point transformToWorld(int \_x, int \_y) throws ExceptionDriver: This function is implemented to get world coordinates of pixel coordinates by using component library functions.

redrawMap() throws ExceptionDriver: This function is implemented to redraw map and layer components by using component library functions.

`double getMapScale()` throws `ExceptionDriver`: This function is implemented to get scale of map component by using component library functions.

`paintMap(java.awt.Color _color)` throws `ExceptionDriver`: This function is implemented to paint map component by using component library functions.

`zoomXTimes(double _times)` throws `ExceptionDriver`: This function is implemented to zoom map by specified times by using component library functions.

`zoomToMapScale(double _scale)` throws `ExceptionDriver`: This function is implemented to zoom map to specified scale by using component library functions.

`addTool(Object _tool)` throws `ExceptionDriver`: This function is implemented to add tool object to map component by using component library functions.

`panToRegionWithDegree(boolean _isLR, int _RN, double _degree)` throws `ExceptionDriver`: This function is implemented to pan map to left/right, south/north with `_degree` by using component library functions.

`setUnit(int _unit)` throws `ExceptionDriver`: This function is implemented to set map component unit by using component library functions.

`returnToPreviousRegion()` throws `ExceptionDriver`: This function is implemented to set map component extent to previous extent by using component library functions.

`returnToInitialRegion()` throws `ExceptionDriver`: This function is implemented to set map component extent to initial extent by using component library functions.

`setCurrentMapExtent(java.awt.Rectangle _region)` throws `ExceptionDriver`: This function is implemented to set map component extent to given region parameter by using component library functions.

`java.awt.Rectangle currentMapExtent()` throws `ExceptionDriver`: This function is implemented to get map component extent by using component library functions.

`java.awt.Rectangle currentMapBounds()` throws `ExceptionDriver`: This function is implemented to get map component bounds by using component library functions.

`addListenerMouseListener(java.awt.event.MouseListener ml)`: This function is implemented to add mouse event listener to map component by using component library functions.

`addListenerMouseMotionListener(java.awt.event.MouseMotionListener ml)`: This function is implemented to add mouse motion event listener to map component by using component library functions.

### **DriverLayer**

**Description:** This is an interface which a driver class written to use layers in component library implements. This interface is extended from Driver interface.

#### **Methods:**

`java.awt.Component getLayerComponent()` throws `ExceptionDriver`: This function is implemented by the driver class implementing this interface. The driver class implements this function by using component library functions.

### **DriverTool**

**Description:** This interface is used to supply requested functions of tool by using component library function and implemented by a class. The map bean at runtime reads the implementing class and functions are invoked.

#### **Methods:**

`boolean getOK(Object _toolClass)` throws `ExceptionDriver`: This function is implemented to finish tool action by using component library functions.

`Object getZoomInTool()` throws `ExceptionDriver`: This function is implemented to get zoom in tool object by using component library functions.

`Object getZoomTool()` throws `ExceptionDriver`: This function is implemented to get zoom tool object by using component library functions.

`Object getZoomOutTool()` throws `ExceptionDriver`: This function is implemented to get zoom out tool object by using component library functions.

`Object getPanTool()` throws `ExceptionDriver`: This function is implemented to get pan tool object by using component library functions.

Object `getDistanceTool()` throws `ExceptionDriver`: This function is implemented to get distance tool object by using component library functions.

### **DriverCoordinateConverter**

**Description:** This interface is used to supply requested functions of coordinate converter class by using component library function and implemented by a class. The map bean at runtime reads the implementing class and functions are invoked.

#### **Methods:**

`Geo2UTM(double _longitude, double _latitude)` throws `ExceptionDriver`: This function is implemented to convert Geographic coordinate to UTM coordinate by using component library functions.

`double getRight()` throws `ExceptionDriver`: This function is implemented to get right part of UTM coordinate by using component library functions.

`double getUp()` throws `ExceptionDriver`: This function is implemented to get up part of UTM coordinate by using component library functions.

`int getRefMeridyen()` throws `ExceptionDriver`: This function is implemented to get reference meridian part of UTM coordinate by using component library functions.

`int getDilimGen()` throws `ExceptionDriver`: This function is implemented to get piece part of UTM coordinate by using component library functions.

`UTM2Geo(double _right, double _up, int _ref_meridyen, int _dilim_gen)` throws `ExceptionDriver`: This function is implemented to convert UTM coordinate to Geographic coordinate by using component library functions.

`double getLatitude()` throws `ExceptionDriver`: This function is implemented to get latitude part of Geographic coordinate by using component library functions.

`double getLongitude()` throws `ExceptionDriver`: This function is implemented to get longitude part of Geographic coordinate by using component library functions.

## **DriverLayerAnalysis**

**Description:** This is an interface which a driver class written to use analysis functions in component library implements. This interface is extended from DriverLayer interface.

### **Methods:**

setFiles(java.util.Vector \_vectorFiles) throws ExceptionDriver: This function is implemented to set files which include elevation values.

boolean doVisibilityAnaysis(java.awt.Point \_point1, java.awt.Point \_point2, double \_scaleX, double \_scaleY, int basAci, int bitAci, double yaricap, int antennaHeight) throws ExceptionDriver: This function is implemented to analyse visibility of a point by using component library functions.

java.util.Vector computeProfile(java.util.Vector \_points) throws ExceptionDriver: This function is implemented to compute profile between given \_points parameter by using component library functions.

int queryHeight(double \_x, double \_y) throws ExceptionDriver : This function is implemented to query height of the point(\_x, \_y) by using component library functions.

double queryDistance(java.awt.Point \_point1, java.awt.Point \_point2) throws ExceptionDriver: This function is implemented to query distance between two points by using component library functions.

## **DriverLayerRaster**

**Description:** This is an interface which a driver class written to use raster layer functions in component library implements. This interface is extended from DriverLayer interface.

### **Methods:**

setMapFile(String \_mapFile): This function is implemented to set raster map file by using component library functions.

## **DriverLayerVector**

### ***Description:***

This is an interface which a driver class written to use vector layer functions in component library implements. This interface is extended from DriverLayer interface.

### ***Methods:***

setMapFile(String \_mapFile): This function is implemented to set vector map file by using component library functions.

## **DriverLayerSymbolsGeometry**

***Description:*** This is an interface which a driver class written to use geometry layer functions in component library implements. This interface is extended from DriverLayer interface.

### ***Methods:***

setColor(java.awt.Color \_color) throws ExceptionDriver: This function is implemented to set color of symbols by using component library functions.

isFiiled(boolean \_isFilled) throws ExceptionDriver: This function is implemented to draw filled symbols by using component library functions.

drawPOINT(int \_x1, int \_y1) throws ExceptionDriver: This function is implemented to draw point symbols by using component library functions.

drawLINE(int \_x1, int \_y1, int \_x2, int \_y2) throws ExceptionDriver: This function is implemented to draw line symbols by using component library functions.

drawPOLYLINE(java.util.Vector \_points) throws ExceptionDriver: This function is implemented to draw polyline symbols by using component library functions.

drawCIRCLE(int \_x, int \_y, int \_radius) throws ExceptionDriver: This function is implemented to draw circle symbols by using component library functions.



`drawELLIPSE(int _x, int _y, int _width, int _height)` throws `ExceptionDriver`: This function is implemented to draw ellipse symbols by using component library functions.

`drawRECTANGLE(int _x1, int _y1, int _x2, int _y2)` throws `ExceptionDriver`: This function is implemented to draw rectangle symbols by using component library functions.

`drawPOLYGON(java.util.Vector _points)` throws `ExceptionDriver`: This function is implemented to draw polygon symbols by using component library functions.

`removeFromList(int _index)` throws `ExceptionDriver`: This function is implemented to draw remove symbols from layer by using component library functions.

### **DriverLayerSymbolsDrawing**

**Description:** This is an interface which a driver class written to use drawing layer functions in component library implements. This interface is extended from `DriverLayerSymbolsGeometry` and `ListenerMouseEvent` interface.

#### **Fields:**

`final static int MODENONE = -1;`

`final static int MODEPOINT = 0;`

`final static int MODELINE = 1;`

`final static int MODEPOLYLINE = 2;`

`final static int MODECIRCLE = 3;`

`final static int MODEELLIPSE = 4;`

`final static int MODERECTANGLE = 5;`

`final static int MODEPOLYGON = 6;`

#### **Methods:**

setDrawingMode(int \_drawmode) throws ExceptionDriver: This function is implemented to set drawing mode to one of the fields above by using component library functions.

clearShapes()throws ExceptionDriver: This function is implemented to clear all drawing geometry symbols from layer by using component library functions.

### **DriverLayerSymbolsFont**

**Description:** This is an interface which a driver class written to use font layer functions in component library implements. This interface is extended from DriverLayer interface.

#### **Methods:**

setFontType(java.awt.Font \_fontType) throws ExceptionDriver: This function is implemented to set font symbols by using component library functions.

changeLocation(int \_index, java.awt.Point point)throws ExceptionDriver: This function is implemented to change location of symbols identified by \_index to point coordinate by using component library functions.

clear(int \_index) throws ExceptionDriver: This function is implemented to remove symbol identified by \_index from layer by using component library functions.

clearAll() throws ExceptionDriver: This function is implemented to remove all symbols by using component library functions.

draw(String \_symbol, java.awt.Point point, java.awt.Color \_color throws ExceptionDriver: This function is implemented to draw symbol identified by \_symbol at point location with color parameter by using component library functions.

changeColor(int \_index, java.awt.Color \_color)throws ExceptionDriver: This function is implemented to change color of symbol identified by \_index by using component library functions.

## **DriverLayerSymbolsImage**

**Description:** This is an interface which a driver class written to use image layer functions in component library implements. This interface is extended from DriverLayer interface.

### **Methods:**

setBorderWidthHeight(int \_width, int \_height) throws ExceptionDriver: This function is implemented to set width and height of image symbols by using component library functions.

clear(int \_index) throws ExceptionDriver: This function is implemented to remove symbol identified by \_index from layer by using component library functions.

clearAll() throws ExceptionDriver: : This function is implemented to remove all symbols by using component library functions.

int draw(java.awt.Image \_image, java.awt.Point point, java.awt.Color \_color) throws ExceptionDriver: This function is implemented to draw symbol identified by \_image at point location with color parameter by using component library functions.

changeLocation(int \_index, java.awt.Point point) throws ExceptionDriver: This function is implemented to change location of symbols identified by \_index to point coordinate by using component library functions.

## **B.6. Beans**

### **BeanMap**

**Description:** Map Bean describes functionalities of map component in this framework. It implements InterfaceMap interface and extended from AbstractBean class that is core class for beans in this framework. A map bean includes a DriverMap object. This object means a driver implementing DriverMap interface by using functions of the selected component library. A map bean uses specified methods in this DriverMap interface to implement the functions declared in InterfaceMap bean.

**Communication with other beans:** A map bean is the core bean in this framework. All other beans are added to the map bean. Beans are added to/ removed

from BeanMap by using addToolBean/ removeToolBean, addLayer/ removeLayer, addCoordinateBean/ removeCoordinateBean.

Beans that want to listen to map bean implement ListenerMap interface to be notified with the events occurred in the bean. Raster Layer and Coordinate Converter beans want to listen to map bean in this framework. Therefore, they implement ListenerMap interface. They register themselves to map bean when they are added to the map bean automatically. In addition, when a new bean that wants to listen to map bean can be registered to map bean by using addListenerMapListener and removed removeListenerMapListener methods. Map bean notifies its listeners when the scale of the map changes and coordinate unit of the map is setted.

A map bean listens to Tool, Raster Layer, Vector Layer, and Analysis Layer and Coordinate Converter beans. Therefore, the map bean implements listeners of these beans, which are ListenerTool, ListenerLayerRaster, ListenerLayerVector, ListenerLayerAnalysis, ListenerCoordinateConverter listener interfaces correspondingly. When these beans are added to the map bean, map bean registers itself to these beans automatically.

## **BeanTool**

**Description:** Tool Bean describes functionalities of tool component in this framework. It implements InterfaceTool interface and extended from AbstractBean class that is core class for beans in this framework. This bean has a thread, which checks the completion of tool bean's work. A tool bean includes a DriverTool object. This object means a driver class implementing DriverTool interface by using functions of a selected component library. A tool bean uses specified methods in this DriverTool object. Tool bean calls these functions in driver class.

**Communication with other beans:** Beans that want to listen to tool bean implement ListenerTool interface to be notified with the events occurred in the bean. These beans subscribe to and unsubscribe from tool bean by using addListenerToolListener and removeListenerToolListener methods correspondingly. Map bean is a listener of tool bean. Therefore, it implements ListenerTool interface. When tool bean is added to map bean, map bean register itself to tool bean automatically.

Tool bean notifies its listeners when zoom, pan, distance and unsetting requests are done from the outside world.

### **BeanLayerAnalysis**

**Description:** Analysis Layer Bean describes functionalities of analysis layer component in this framework. It implements InterfaceLayerAnalysis interface and extended from AbstractBeanLayer class that is core class for layer beans in this framework. An analysis layer bean includes a DriverLayerAnalysis object. This object means a driver implementing DriverLayerAnalysis interface by using functions of a component library. An analysis layer bean uses specified methods in this DriverLayerAnalysis object. Analysis Layer bean calls these functions in driver class.

**Communication with other beans:** Beans that want to listen to analysis layer bean implement ListenerLayerAnalysis interface to be notified with the events occurred in the bean. These beans subscribe to and unsubscribe from tool bean by using addListenerLayerAnalysisListener and removeListenerLayerAnalysisListener methods correspondingly. Map bean is a listener of analysis layer bean. Therefore, it implements ListenerLayerAnalysis interface. When analysis layer bean is added to map bean, map bean register itself to analysis layer bean automatically.

Analysis layer bean notifies its listeners when a pixel coordinate is selected by a mouse click in the outside world to request conversion of these pixels to world coordinates.

### **BeanLayerRaster**

**Description:** Raster Layer Bean describes functionalities of raster layer component in this framework. It implements InterfaceLayerRaster interface and extended from AbstractBeanLayer class that is core class for layer beans in this framework. A raster layer bean includes a DriverLayerRaster object. This object means a driver implementing DriverLayerRaster interface by using functions of a component library. A raster layer bean uses specified methods in this DriverLayerRaster object. Raster Layer bean calls these functions in driver class.

Raster Layer bean manages showing raster layers according to the scale of the map. For example if the scale of the map is 375000, raster maps with scale of 1/500000 are shown.

**Communication with other beans:** Beans that want to listen to raster layer bean implement `ListenerLayerRaster` interface to be notified with the events occurred in the bean. These beans subscribe to and unsubscribe from tool bean by using `addListenerLayerRasterListener` and `removeListenerLayerRasterListener` methods correspondingly. Map bean is a listener of raster layer bean. Therefore, it implements `ListenerLayerRaster` interface. Raster Layer bean is a listener of map bean in this framework. When raster layer bean is added to map bean, raster layer bean register itself to map bean and map bean register itself to analysis layer bean automatically.

Raster layer bean notifies its listeners when visibility of raster map is set from outside, and raster maps are loaded according to the new scale.

### **BeanLayerVector**

**Description:** Vector Layer Bean describes functionalities of vector layer component in this framework. It implements `InterfaceLayerVector` interface and extended from `AbstractBeanLayer` class that is core class for layer beans in this framework. A vector layer bean includes a `DriverLayerVector` object. This object means a driver implementing `DriverLayerVector` interface by using functions of a component library. A vector layer bean uses specified methods in this `DriverLayerVector` object. Vector Layer bean calls these functions in driver class.

**Communication with other beans:** Beans that want to listen to vector layer bean implement `ListenerLayerVector` interface to be notified with the events occurred in the bean. These beans subscribe to and unsubscribe from tool bean by using `addListenerLayerVectorListener` and `removeListenerLayerVectorListener` methods correspondingly. Map bean is a listener of vector layer bean. Therefore, it implements `ListenerLayerVector` interface. When vector layer bean is added to map bean, vector layer bean register itself to map bean automatically.

Vector layer bean notifies its listeners when visibility of vector map is set from outside.

### **BeanLayerSymbolsDrawing**

**Description:** Symbols Drawing Layer Bean describes functionalities of drawing layer component in this framework. It implements `InterfaceLayerSymbolsDrawing` interface and extended from `BeanLayerSymbolsGeometry` class framework. A

drawing layer bean includes a DriverLayerSymbolsDrawing object. This object means a driver implementing DriverLayerSymbolsDrawing interface by using functions of a component library. A raster layer bean uses specified methods in this DriverLayerSymbolsDrawing object. Drawing Layer bean calls these functions in driver class.

### **BeanLayerSymbolsGeometry**

Description: Geometry Layer Bean describes functionalities of geometry layer component in this framework. It implements InterfaceLayerSymbolsGeometry interface and extended from AbstractBeanLayerSymbols class that is core class for layer beans on which symbols are shown in this framework. A geometry layer bean includes a DriverLayerSymbolsGeometry object. This object means a driver implementing DriverLayerSymbolsGeometry interface by using functions of a component library. A geometry layer bean uses specified methods in this DriverLayerSymbolsGeometry object. Raster Layer bean calls these functions in driver class.

### **BeanLayerSymbolsFont**

*Description:* Font Symbols Layer Bean describes functionalities of font symbols layer component in this framework. It implements InterfaceLayerSymbolsFont interface and extended from AbstractBeanLayerSymbols class that is core class for layer beans on which symbols are shown in this framework. A font layer bean includes a DriverLayerSymbolsFont object. This object means a driver implementing DriverLayerSymbolsFont interface by using functions of a component library. A font layer bean uses specified methods in this DriverLayerSymbolsFont object. Font symbols layer bean calls these functions in driver class.

### **BeanLayerSymbolsImage**

*Description:* Image Symbols Layer Bean describes functionalities of image symbols layer component in this framework. It implements InterfaceLayerSymbolsFont interface and extended from AbstractBeanLayerSymbols class that is core class for layer beans on which symbols are shown in this framework. An image layer bean includes a DriverLayerSymbolsImage object. This object means a driver implementing DriverLayerSymbolsImage interface by using functions of a

component library. An image layer bean uses specified methods in this DriverLayerSymbolsImage object. Image symbols layer bean calls these functions in driver class.

### **BeanCoordinateConverter**

**Description:** Coordinate converter bean describes functionalities of coordinate converter component in this framework. It implements InterfaceCoordinateConverter interface and extended from AbstractBean class that is core class for beans in this framework. A coordinate converter bean includes a DriverCoordinateConverter object. This object means a driver implementing DriverCoordinateConverter interface by using functions of a component library. A coordinate converter bean uses specified methods in this DriverCoordinateConverter object.

**Communication with other beans:** Beans that want to listen to coordinate converter bean implement ListenerCoordinateConverter interface to be notified with the events occurred in the bean. These beans subscribe to and unsubscribe from tool bean by using addListenerCoordinateConverterListener and removeListenerCoordinateConverterListener methods correspondingly. Map bean is a listener of coordinate converter bean. Therefore, it implements ListenerLayerRaster interface. Coordinate converter bean is a listener of map bean in this framework. When coordinate converter bean is added to map bean, coordinate converter bean register itself to map bean and map bean register itself to coordinate converter bean automatically.

Coordinate converter bean notifies its listeners when request for conversion of pixel coordinates of a selected point into the requested coordinate system is done.

### **BeanLayerSymbolsFontTracking**

**Description:** Tracking Font Symbols Layer Bean describes functionalities of tracking font symbols layer component in this framework. It is extended from BeanLayerSymbolsFont class that is a layer bean on which font symbols are shown. Font symbols on this layer move in a period of time and implements InterfaceBeanLayerSymbolsTracking interface. Therefore, this bean has a function that is used to set this time interval.



### **BeanLayerSymbolsGeometryTracking**

**Description:** Tracking Geometry Symbols Layer Bean describes functionalities of tracking geometry symbols layer component in this framework. It is extended from BeanLayerSymbolsGeometry class that is a layer bean on which geometry symbols are shown and implements InterfaceBeanLayerSymbolsTracking interface. Geometric symbols on this layer move in a period of time. Therefore, this bean has a function that is used to set this time interval.

### **BeanLayerSymbolsImageTracking**

**Description:** Tracking Image Symbols Layer Bean describes functionalities of tracking image symbols layer component in this framework. It is extended from BeanLayerSymbolsImage class that is a layer bean on which image symbols are shown and implements InterfaceBeanLayerSymbolsTracking interface. Image symbols on this layer moves in a period of time. Therefore, this bean has a function that is used to set this time interval.