Master Thesis
Computer Science
Thesis no: MCS-2007-19
September 2007

Modeling Patterns in Software Design and Use Cases

Department of
Interaction and System Design
School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby

This thesis is submitted to the Department of Interaction and System Design, School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 24 weeks of full time studies.

Contact Information:
Author(s):
Ahmad Waqas
Fawad Kamal
Address:
E-mail: {awka05, afka06}@student.bth.se

University advisor(s):
Hans Kyhlback Email: hans.kyhlback@bth.se
Department of Interaction and System Design

Department of
Interaction and System Design
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax: + 46 457 102 45

Acknowledgement

"All that I am or ever hope to be, I owe to my angel Mother".

(Abraham Lincoln)

Abstract

Software patterns provide solutions to recurring design problems, provide a way to reason about the quality attributes, and support stakeholders in understanding the system. Naturally, the use of software patterns emerges from the requirements of the software. Use Cases have been a traditional and authentic approach to document these requirements. We propose a way to mine these patterns by using use cases and advocate their significance in saving resources and time. For this purpose, an open-source system is discussed and four design patterns are mined with the help of use cases elicited from the documentation and literature available for the selected system. Patterns mined in this system are then document in GOF format. Furthermore, the consequences of few patterns on quality attributes are studied and an additional design pattern is proposed to improve the quality of the system.

Keywords:
Software Patterns, Quality Attributes, Use Cases

TABLE OF CONTENTS

1 FIGURES

2 TABLES

3 INTRODUCTION

Architectural [5] and design patterns [4] are a proven technique to design software systems with high quality. These patterns help software architects to reason about the functionality and quality of the system without going into the detail level implementation. These patterns provide solutions to recurring problems, encountered during past experiences, in developing well-structured and quick solutions.

In current era, software patterns are widely used in developing projects varying from small scale to large scale which opens another era of discussion in finding patterns in existing system. This is due to the reason that a large scale of software development takes place in building blocks of existing system. However, patterns demand careful selection and implementation approach, neglecting of which can lead to bad design and poor quality of the system. It is important to not only analyze the individual support offered by software patterns but to analyze there impact on software system when a combination of patterns is used in the system. So, the architects can benefit in arguing about the quality of the architecture of an existing system by identifying the combined effect of the patterns used in the system. It is due to the reason that it is possible that quality characteristics of one pattern may be affected by the use of another pattern in the system. For instance, Layers [5] are good for maintainability but can affect badly the performance of the system. On the other hand, Broker [5] pattern can affect maintainability negatively but it is considered good for the security of the system. If handled carelessly, the patterns can be costly to implement and can possibly increase the complexity of software projects. Keeping in view the above discussion we define the difference between architectural patterns, design patterns and idioms which will help reader to understand our work in the rest of the document:

> **Architectural Pattern –** used at an abstract level of system design i.e. use of components to reflect the structure of system.
>
> **Design Patterns –** used under the cover of abstraction, these are more close to source code but in actual does not contain any source code i.e. relationships among classes can reveal a design patterns.
>
> **Idioms –** Work at the detail level implementation, patterns identified at the source code level are more often recognized as idioms.

Requirements generation through use cases has been an active area of work in software industry. A lot of research has already been done in eliciting requirements using use cases e.g. eliciting efficiency requirements with use cases [11], Eliciting and specifying requirements with use cases [12] etc. We advocate the use cases with their use in architecture design of the system as well i.e. realization of software patterns in the architecture of the system. We consider use cases as a combination of one or more than one scenarios that provide alternative ways of using the system. These use cases in essence define the functionality of the system which can lead us to the existence of patterns modeled in the system. Our work further finds the importance of patterns in achieving the quality attributes of the system and the way patterns help understand the structure of the system.

In this paper we discuss few specific patterns and illustrate their consequences on few specific quality attributes of the selected system. Each software pattern inherits specific quality attributes that are influenced both positively and negatively by other patterns used in the system. We discuss a specific

quality attribute of Flexibility and reason about the patterns associated to this quality attribute with the help of an example system design. Furthermore, we discuss an open-source system, study its design at both abstract, and detail level to mine four design patterns. In both of the example projects, we specifically discuss few specific quality attributes that have consequences on the patterns used in these systems.

The report is structured as follows: In section 4, we provide some motivation about using use cases to mine patterns. In section 5, we illustrate use cases, the way use cases help document the system, and the way use cases relate to each other. In section 6, we briefly elaborate with the help of an example system, the linking between patterns and quality attribute of flexibility. Both the sections 5 and 6 build-up the basis and readers understanding about use cases, patterns, and quality attributes. In section 7, an example Eclipse system is studied to mine and document four patterns with the help of use cases. In section 8, a brief conclusion to this study is presented.

4 MOTIVATION

Software patterns let software engineers to reason about the quality attributes of the system. A large number of new applications are designed with the use of software patterns in the system. However, the challenging task is to find patterns in systems that have been in place for years. The situation is worsened when systems are poorly documented and haphazardly developed. In such case one solution used by software engineers, who are assigned the task to re-engineer the system, is to study the source code of the system to understand its implementation and to find the used patterns. We are not against this approach, however we suggest another approach introduced in this section due to following reasons:

1. Architecture patterns provide general structure of the applications as solution to recurring problems; these do not deal with detail level implementation of the system at source code level, easing the requirement for the expertise to understand complex code.

2. Design patterns are at more detail level but these too are not always implemented at source code level rather they provide intermediary structure of the system.

3. Source code analysis of the system may result in un-necessary clues of low-level patterns that may not be beneficial for the overall quality of the system.

4. Source code study may be good to find idioms, a detail level implementation of patterns, but not for higher level design patterns.

5. It may become a too time consuming activity to study the complete source code of the system.

Keeping in view the above highlighted points, we propose a way to utilize use cases as a clue leading to patterns to be mined in the system. We believe that this strategy not only helps to document existing software patterns but it saves resources and time as well. We expect that after reading this report, readers will be able to get in-depth knowledge about the use of patterns in the system and the way these patterns affect quality attributes of the system. We emphasis the use of use cases because we consider them as one popular and well-known strategy to define and document the requirements and functionality of the system. When use cases are already well written in system, our approach can benefit in a better way. Further, by relating a group of use cases to a particular pattern helps software engineer to reason about the portion of functionality directly affected by the patterns. Once a list of such use cases are identified that directly map to the presence of patterns in the system; the source code portions of the system only affected by the particular use cases can be identified. This strategy saves the time by limiting the focus to specific portions of application, instead of making a complete walk through of the whole application source code.

5 USE CASE MODELING AND THEIR RELATIONSHIP TO SOFTWARE ARCHITECTURE

In this chapter we discuss the contents of use cases and architecture patterns, while in the subsequent sections we discuss how these patterns and use cases can help in identification and validation of quality attributes of the system. We consider it important to highlight the notions of use cases and architecture patterns in this section so that familiarity of reader with both of these concepts is matured enough to understand our work in next sections. We will discuss different sections of use case design, architecture patterns and their importance in architecture design.

5.1 USE CASES AND THEIR RELATIONSHIPS

A use case describes a particular action on the system and behavior of the system under different conditions and circumstances [13]. Person who performs this action on the system is known as actor or user of the system who is responsible to invoke and execute a use case on the system. A use case can represent any event occurring in our daily life, like action to purchase a ticket, online shopping, patient examination etc. In the 'software world' use cases can be mapped to any phase of software development life cycle like requirement specification, software design, development and testing. In this way use cases can be defined as 'use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal' [15]. However this definition does need more explanation regarding connection between actor and system. Also in real systems it is normally the case that finish point of one use case may be used as a start point of some other use case. Definition described above define use cases as a set of sequences, these sequences can have their scope to more than one actors or system. We define actor as an entity who invokes an action to achieve a goal and to fulfill this goal it needs to perform a set of actions. These actors can be internal or external to the system. An internal actor corresponds to system under design, modules or objects while an external actor can be a person, another system or hardware etc. We describe each use case with a set of associated goals. These goals can be mentioned in any form; traditionally these were the functional statement part of the use case or objective of the use case. If required we can define backup actions for goals. These backup actions work similar to the famous algorithmic if, else statements i.e. if second actor does not deliver its responsibility then primary actor has to find another way to fulfill its goal [13].

Interaction among actors and system could be simple or compound. An interaction like 'Show Patient Chart' is a simple interaction with the system while an interaction like 'Patients scheduled with PhysicianA should be displayed in the list of PhysiacianB in brown color' represents a compound level of interaction. Clearly the pre-requisite to this use case is scheduling of patients accurately. Also, this use case shows interaction of sequences like patient scheduling, show patients to second physician and display of patients in different color. Such communication can be simple or it can have a nested loop to execute a multiple set of sequences. This interaction is to show that at higher level a use case represents input and output of values but at detail level it may have different interaction levels. It is better not to go into the details of these interaction levels at higher level to reduce complexity and to safe time. Like a use case 'Payment' can contain different input values like payment by cash, check, credit, credit card,

e-transfer etc. In the detail level of use case each of these inputs may follow a different set of interaction level depicting nested loops [13].

5.2 DOCUMENTING SOFTWARE SYSTEMS IN USE CASES

The traditional approach is to document the requirements specified by the stakeholders either in natural language or well defined format; such as requirements numbering in bullets and section format. Both of these techniques were successful and still used in industry. But analysts realized that a lot of time is wasted and cost is involved when developers try to map the requirements to their development code or program. Many of the requirements specified in specification were found missing, assumed, ambiguous or unclear. At that time use cases were merely thought of as an alternate way to specify requirements, it was thought that natural language and traditional techniques can always be used irrespective of use cases and use cases can only be used for specific projects not all projects. Later analysts realized that projects in which use cases are used are more successful in terms of development than projects without use cases, the development team also showed their concern for the use cases because requirement specification done with the use cases were much clear, complete and consistent. Use case approach gave the clear view about what customer wants from the project and automated system from developer, it provided the complete scenario and made clear boundaries on functional requirements of the system. So developers don't have to make unnecessary assumptions, it will be easy for the development team to map the requirements straight way to their program. Latest trend to reflect these requirements in software architecture is use of architecture patterns. These architecture patterns exhibit specific quality attributes that can clearly specify the nature of the system. For example calling a system as layered gives a picture of a system that is easily maintainable, a pipe-filter pattern gives computational image of system and a client-server system reflects a system with communication network. Our work captures the area of architecture design by combining use cases and architecture patterns. We feel that both of these important concepts are independently worked in industry but they can be combined to get extra advantage in modeling of software architecture.

Also, use cases changed the vision of requirement engineering practices. It is the most efficient and effective technique that is adopted by the organizations now a days for the software development. Objective of the use-case approach is to describe all tasks that users will need to carry out with the system, though in reality it will not describe the entire task but it will bring the requirements more clearly and closer than it was before while using traditional approaches [9].

Use of use cases in requirement engineering not only helps requirement engineers to better map their requirements to different development phases but it also provides a better way of understanding the requirements [10, pp.17]. Like, use cases mapping to design, development and testing and easy understanding of system functionality by actor system relationship defined by use cases. We consider software architecture as the potential area of resource consumption in the development life cycle for ever changing requirements.

An example of use case description is shown in Fig. 1:

Use Case ID: PID_MODID_027

Pre-Req. Use Cases: PID MODID 017

Use Case Purpose: To verify that patients scheduled by secretary are displayed in associated Physician list.

Actors:
Secretary: Schedules a patient for physician
Physician: Examines the patients scheduled by secretary

Use Case: Patient Scheduling

Invocation Conditions: Secretary has successfully logged to the system and has access to scheduling module.

Flow Conditions:
1. Valid patient is registered into the system
2. Empty slot is available to schedule the patient.

Flow of Events:
1. Secretary accesses the scheduling module and selects the physician for which she wants to schedule the patient.
2. Secretary selects an empty slot for an appointment for the patient.
3. On selection of slot 'New Appointment' screen is displayed.
4. Secretary searches the registered patient data and makes sure that all mandatory fields are filled.
5. For the scheduled patient, secretary checks its association with any insurance company and if available provides insurance data.
6. Secretary saves the patient data and patient is moved to Physician list in the allocated slot.

Figure 1: Use Case Description [11, pp.63]

Also every system has some interacting units that interact with other entities to produce or compile the results. Actors that interact with the system can be humans or other systems that may invoke some operation. Thus use cases describe the behavior of the system or a part of the system and describe a set of sequence of actions that a system performs to produce results for an actor. In this way use cases involve the interaction of actors and system. An actor represents a logical set of roles that users of the use case perform when interacting with the use case.

Use case contents can vary from organization to organization or even project to project within an organization. Best use cases are those that meet the organization and project's requirements. Organizational requirements reflect rules, policies and constraints while project requirements mostly represent functional requirements. It is necessary that use cases are designed in a way that they fulfill both of these requirements.

Use case development allows requirement engineers to establish a relationship between functional and non-functional requirements of the intended system and the organizational goals. This is due to the reason that irrespective of the application goals, organizations set their overall goals as well. These organizational goals can be mapped to the use cases goals [13], which can help requirement engineers to eliminate the requirements that are un-necessary and never used in implementation.

There are technical benefits too. One of the popular ways of software development these days is to use object oriented approach. As use cases depict domain objects of the application, so developers can directly turn use cases into object models such as class and sequence diagrams. Combinations of these objects, classes and diagrams can actually be represented in architecture patterns as well. However, it is possible that more than one use cases can reflect one object in class or sequence diagram and vice versa. This also provides a mapping between customer's voice (use cases) and actual software implementation.

5.3 USE CASES AND SOFTWARE ARCHITECTURE OF THE SYSTEM

Software architecture provides a high level design of the system that strives to provide different views of the system to its intended audience. Current notion in design of software architecture is to model software patterns that build the structure of software architecture. These architecture patterns are solution to recurring problems in the system and provide an opportunity to re-use tested solutions of known problems. In actual these patterns can directly relate to the requirements of the system. We investigate implementation of these requirements through use cases and then their link to patterns.

5.4 USE CASES AND SCENARIOS

A use case describes a set of sequences, not just a single sequence and it would be impossible to reflect complete behavior of use case in one sequence. Typically a use case covers one main flow and many possible variants to this main flow. Each of these variants reflects a unique sequence of operations also known as scenario of the system whereas each such scenario covers at least one test case in the system. Thus we can illustrate that each system can have dozens of use cases and each use case can involve many scenarios [7, pp190-191].

Off course there is no need to include a section in use case that seems to be vague or that doesn't fulfill your needs. In this thesis we have used use case model followed by Alistair [13]. Also use cases have variants too i.e. use cases that use parts of other system. We can define these variants by three rules i.e. generalization, include and extends. These variants are used to reflect use of use cases over the whole system or only a part of system. Nevertheless in each case, use cases can be used to effectively demonstrate the behavior of the system. [7, pp.185-186]. More detailed explanation about use case variants is provided in next section of organizing use cases.

5.5 ORGANIZING USE CASES

Use cases can be used to reflect behavior of system, sub-system or a part of the system. In this view use cases need to be represented in an organized way, where collaboration and interaction among use cases must be visible. We define three kinds of relationships to define interactions among use cases i.e. generalization, include and extends.

5.5.1 GENERALIZATION

Generalization in use cases is similar to inheritance in object oriented programming languages. Here it reflects that child use case inherits the behavior of parent use case. Similar to object oriented programming languages, this child use case can further define its behavior and functionality. For example for the functionality validate user, it can have further two parts i.e. check password and retinal scan, both of these not only use the behavior of function validate user but carry their own behavior as well. Generalization among use cases is reflected with a solid line with a long arrow head as shown in Fig. 2 [7, pp-192].

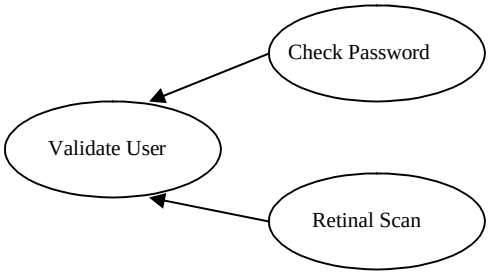


Figure 2: Use Case Generalization relation [7, pp.193]

5.5.2 INCLUDE

An include relationship between use cases indicate that one use case incorporates the behavior of another use case. An included use case never stands alone but it is part of a larger base that includes it. We use include relationship to avoid repetition of some flow of scenario in use cases [7, pp-193].

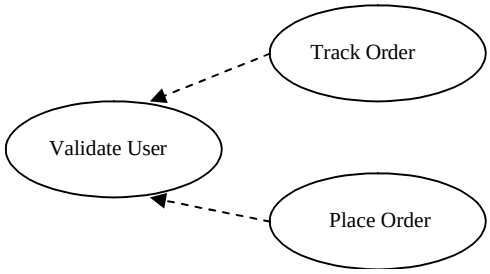


Figure 3: Use Case Include relation [7, pp.193]

An example of include relationship can be demonstrated by the use of following scenario for a use case. Consider different modules of health care software system i.e. registration, scheduling, and patient examination. In order for a system to be examined by physician, he needs to be registered and scheduled for an appointment in the system. Typically if patient examination module consists of a number of use

cases then it is not necessary to repeat steps for registration and scheduling for each use case. Rather these steps can be included in the patient examination module use case as a pre-requisite.

5.5.3 EXTENDS

Extend relationship among use cases show extra functionality of the system. In this way we separate optional behavior from mandatory behavior. Like a use case mandatory behavior could be to schedule a patient for a physician while patient chart display can be an extended behavior of the system.

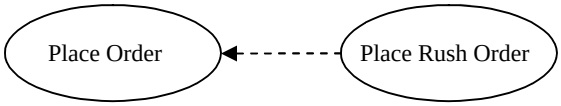


Fig. 4: Use Case Extends Relation [7, pp.193]

Fig.4 shows another example of extend relationship where place order is a routine functionality of the system while place rush order is an extended functionality that shows extra behavior of the system. This distinction among use cases with the help of extend relationship can be useful for other phases of software development life cycle as well. Like during maintenance phase after every change in system, testers can run scenarios related to main use cases and can exclude the extended use cases of the system.

It is recommended to follow standard formats to write use cases and place these use cases in central data repository where these can be used as an input to any phase of software development life cycle. These standards could be defined within organization or can be imported from external environment which constitute the main body of the system by defining contents within the system, use case design patterns, use case format etc. Well documented use cases are easy to read and understand, map system requirements and are flexible to be referenced by developers, architects, testers and different stakeholders of the system.

6 SOFTWARE PATTERNS AND QUALITY ATTRIBUTES

A number of architectural and design patterns exist in literature, and each of these patterns have consequences on specific quality attributes of the system. In this section, we first briefly describe an example system, select quality attribute of flexibility and list a number of architectural and design patterns that have consequences on this quality attribute. We expect that such a study will help readers understand the essence of software patterns and their relation to quality attributes. A vast world of software patterns exist, that can be studied in a similar way to understand other quality attributes and patterns not referred in this section. Further, it is not our purpose to go into detail of these patterns; rather we introduce use to these patterns here and leave the rest of the study to the user.

6.1 DEFINING QUALITY ATTRIBUTES

Quality attributes of a system are mainly non-functional characteristics of the software system that are demanded on the system for its effective work. For instance performance in missile systems, security in defense related software, safety in health care software and maintainability in distributed systems are few examples of quality attributes. It is important to know the quality attributes of the software system at the time of writing software requirements as these quality attributes largely affect the design of software system. In this view, it is important to understand the behavior of quality attributes that relate to software systems. These quality attributes are divided in two types; one that relate to user aspect of software systems i.e. performance, security, availability, usability, interoperability; and the one that are directly concerned with the developers i.e. modifiability, portability, reusability, integrability, testability, flexibility etc. Off course this list is not complete and there is a wide pool of quality attributes available in literature that can be found at SEI [16], Gang of Four [4] and POSA [2] [5] etc. In our work quality attributes play a vital role in the use of software patterns. In the coming sections we see how software patterns are related to quality attributes and how the use of more than one software patterns in the system influences the quality attributed of the system.

6.2 THE FLEXITY ATTRIBUTE AND THE CONNECTOR PATTERN EXAMPLE

The sample example presented in this section is a distributed networked system, which decouples connection establishment from service processing. Many network services, such as remote function calls, file transfer and dynamic web page generation use connection-oriented protocols, such as TCP to transfer data reliably between communication end points [1]. We analyze in this example system how our quality attribute of flexibility is affected both positively and negatively by the combination of few selected patterns. Two important concepts used in this system are gateway and peers as described below:

Peers are located throughout the distributed system network using both local area networks (LANs) and wide area network (WANs). These peers are used to control network resources, such as satellites, call centers or remote branch offices [1].

Gateway: A Gateway works as a mediator between peers for the flow of data. In this example gateways use a connection-oriented inter process communication (IPC) mechanism to transmit data between connected peers [1].

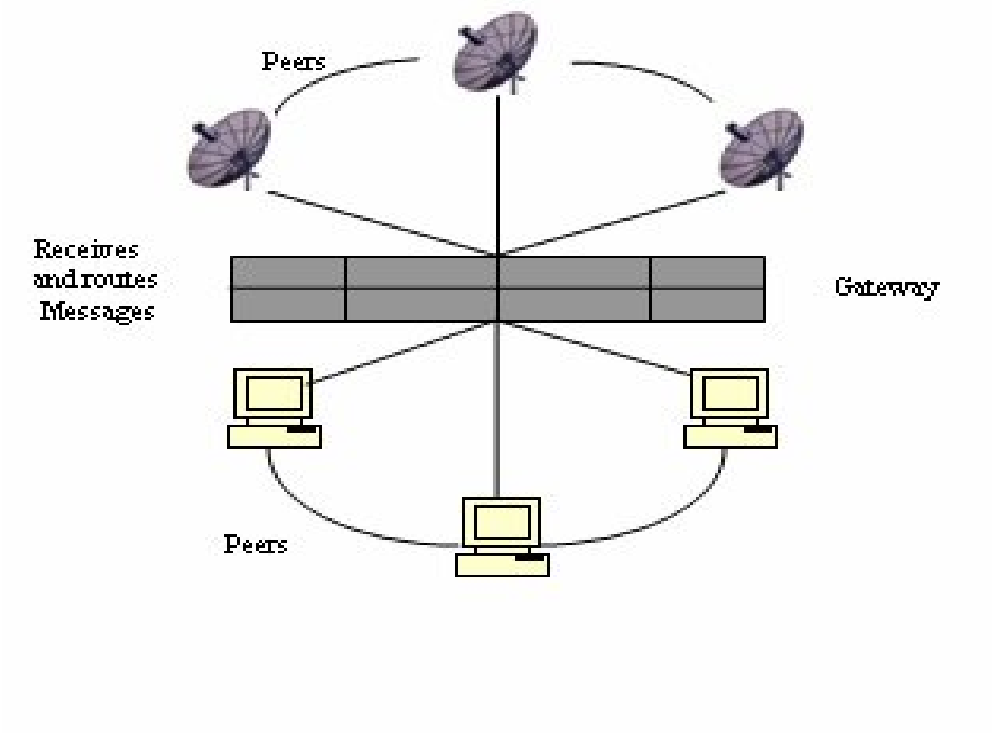


Figure 5: System Implementation using Gateways and Peers

Figure 5 illustrates how gateways and peers are used in the system and figure 6 illustrates participants in the acceptor-connector pattern [2] by illustrating a sample scenario running in the system. The structure in Figure 6 is divided in three layers: Reactive layer, connection layer and application layers. The acceptor and connector components assemble resources and activate handlers which are responsible to exchange data between peers. Components in connection layer can use reactor pattern. For example, the connector's asynchronous initialization strategy establishes a connection after a reactor invokes it, this reactor pattern also enables multiple handlers to be initiated asynchronously [1].

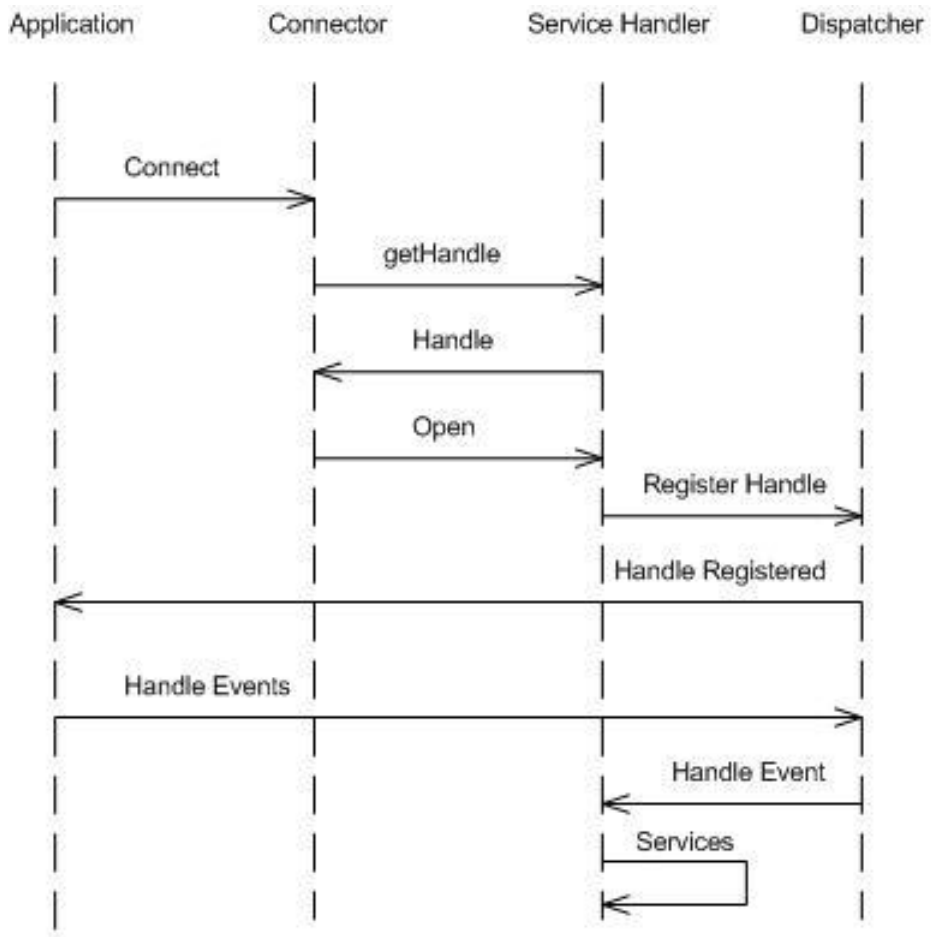


Figure 6: A dynamic scenario of request processing

Scenario processing in figure 6 is based on acceptor-connector pattern, which uses other sub-patterns to complete its implementation. Each request is first initialized by establishing a connection to a network address. A handle is used to establish the connection bypassing the application. The acceptor pattern is used to handle these requests in the system e.g. event handling methods. Furthermore layers are used to separate the application, connection and event handling services [1]. Once the connection is established, all the incoming events coming from application are processed. For all this processing, the system uses patterns e.g. strategy, abstract factory and factory method.

6.3 PATTERNS THAT AFFECT FLEXIBILITY

We define flexibility in distributed networked systems as the characteristics of allowing flexible routing strategies, error handling if connections fail, loose coupling between peers and gateways and flexible connection roles. Table 1 illustrates the patterns that relate to flexibility both positively and negatively in the context of the distributed communication system discussed in previous section.

A: Architecture Pattern
D: Design Pattern

Pattern Name	Pattern Type	Influence	Discussion
Wrapper Façade	Design	+	Specifies underlying mechanism of communication
Abstract Factory	Design	+	Decouples connections
Factory Method	Design	+	Dynamic routing strategies
Strategy	Design	+	A combination of routing strategies can be used
Bridge	Design	+	Loose coupling between peers
Layers	Architecture	+/-	Improve error handling by decoupling functionality
Explicit Invocation	Design	+/-	Improves routing as the topology of interacting components is known, improves error handling
Broker	Architecture	+ / -	Improves routing strategies, however error handling can be effected by strong coupling

Table 1: Software Patterns with Consequences on Quality Attributes

6.4 PATTERNS WITH POSITIVE INFLUENCE ON FLEXIBILITY & CONSEQUENCES ON OTHER QUALITY ATTRIBUTES

The example illustrated above uses the patterns of Wrapper Façade, Factory Method, Strategy, Bridge and Abstract factory to fully implement acceptor-connector pattern. In this section we discuss only the design patterns of Abstract factory, Bridge and Strategy to discuss their implementation and consequences on the quality of the system.

Strategy pattern provide a family of algorithms, encapsulates them and make them interchangeable [3]. In our example the acceptor-connector pattern uses strategy pattern to create, accept and execute handlers [1]. Implementation of this pattern in the system allows choosing initialization strategy among a number of strategies without modifying an algorithm. For instance, the application uses strategies of reactive strategy, where all handlers are executed within same thread; thread strategy, where each handler uses its own thread; thread pool strategy, where each handler is executed within a pool of threads; and process strategy, where each handler is executed in a separate process [1]. When the acceptor is initialized, the strategy factory configures the designated strategy, allowing the selection of algorithm among a family of strategies. This way any of the implemented strategy can be modified, replaced or new strategy can be added which improves the flexibility and maintainability of the system. However, the implementation overhead introduced by strategy pattern can affect the performance of the system negatively.

Abstract Factory pattern provides interface to a family of related objects without specifying their concrete classes [3]. In this example, the acceptor-connector pattern used abstract factory pattern to provide interface to initialization strategies discussed in the previous section i.e. reactive strategy, thread strategy, process strategy and thread pool strategy [1]. This way all the operations are performed through single interface where implementation details are encapsulated from external environment. This adds flexibility in the implementation of different strategies used in the system, however this can affect flexibility negatively when new kind of services need to be introduced in the system.

Factory Method provides interface to an object, but leaves instantiation decision to sub-classes [3]. The acceptor-connector pattern in this example uses this pattern to allow each initialization strategy to be extended without affecting the acceptor or handler implementation [1]. This strategy provides a hook to sub-classes so, that these can be further extended. This allows creating objects inside a class rather than creating an object directly [3].

The three patterns of Strategy, Abstract Factory and Factory Method discussed above mainly influenced the flexibility of the system positively; however these patterns have consequences on the other quality attributes of the system. All three of the patterns influence performance of the system negatively due to the additional communication and implementation overhead in the system; however maintainability of the system is positively influenced mainly due to separation of concrete implementation. Table 2 illustrates the quality attributes influenced by these patterns:

QA Pattern	Flexibility	Performance	Maintainability	Security	Reliability
Strategy	+	-	+	-	+
Abstract Factory	+/-	-	+	+	-
Factory Method	+	-	+	+	+

Table 2: Patterns and related quality attributes in the system

7 AN EXAMPLE SYSTEM OF ECLIPSE MODELING FRAMEWORK

Eclipse modeling framework (EMF) is an open source java based framework that provides a platform for tool development and integration [3] [6]. EMF can work as an intermediary between XML, UML and JAVA by generating source code from one of the language to the other language and by generating new models. It offers a simple approach to model and meta-model driven design of applications with multiple views of java enabled implementation, XML based schemas and UML support [3] [14]. Further, it saves development effort by providing inherent support of code that is repeatedly used in the system and provides a base platform to build the structure of the system. This base platform can be extended to implement the specific needs of the application without affecting the core structure of EMF.

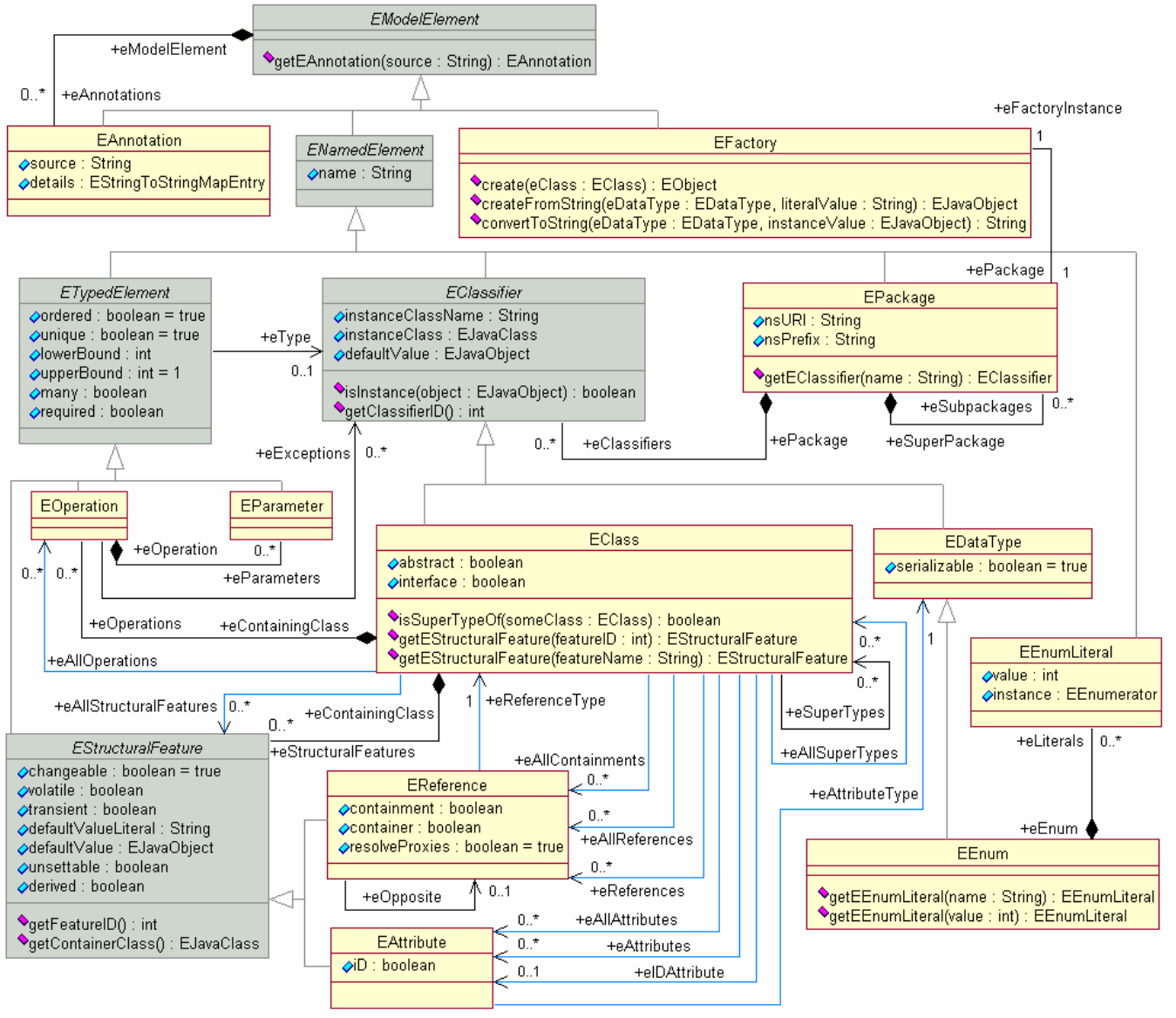


Figure 7: Eclipse Modeling Framework Class Model

Following is a list of few specific classes from EMF class model along with their short descriptions:

EObject is the base interface class that extends Notification class.

Eclass represents the modeled class and specifies attributes and references.

Eattribute contains simple data represented by EdataType.

Ereference represents an association between classes. Once instances of different classes are created, Ereference can be used to link instances of these classes.

Epackage contains information about model classes (Eclass) and their data types (EdataType).

Efactory contains methods to create model elements.

Figures 7 and 8 illustrate the structure of classes used in EMF model. In depth discussion and analysis of every class represented in this model is out of the scope of this work. Instead, we illustrate specific definitions of few classes in this model, with respect to their use in our pattern mining work in next sub-sections. For the purpose of documenting discovered patterns, each of the four patterns mentioned in the sub-sections (7.3, 7.4, 7.5 and 7.6) of this chapter will constitute pattern definition, problem statement, use cases related to the pattern, structure of pattern in EMF, pattern implementation in EMF, and consequences of the pattern on EMF system. Furthermore, we will use the standard use case template, as we discussed in chapter 5 to document the use cases discovered in EMF project.

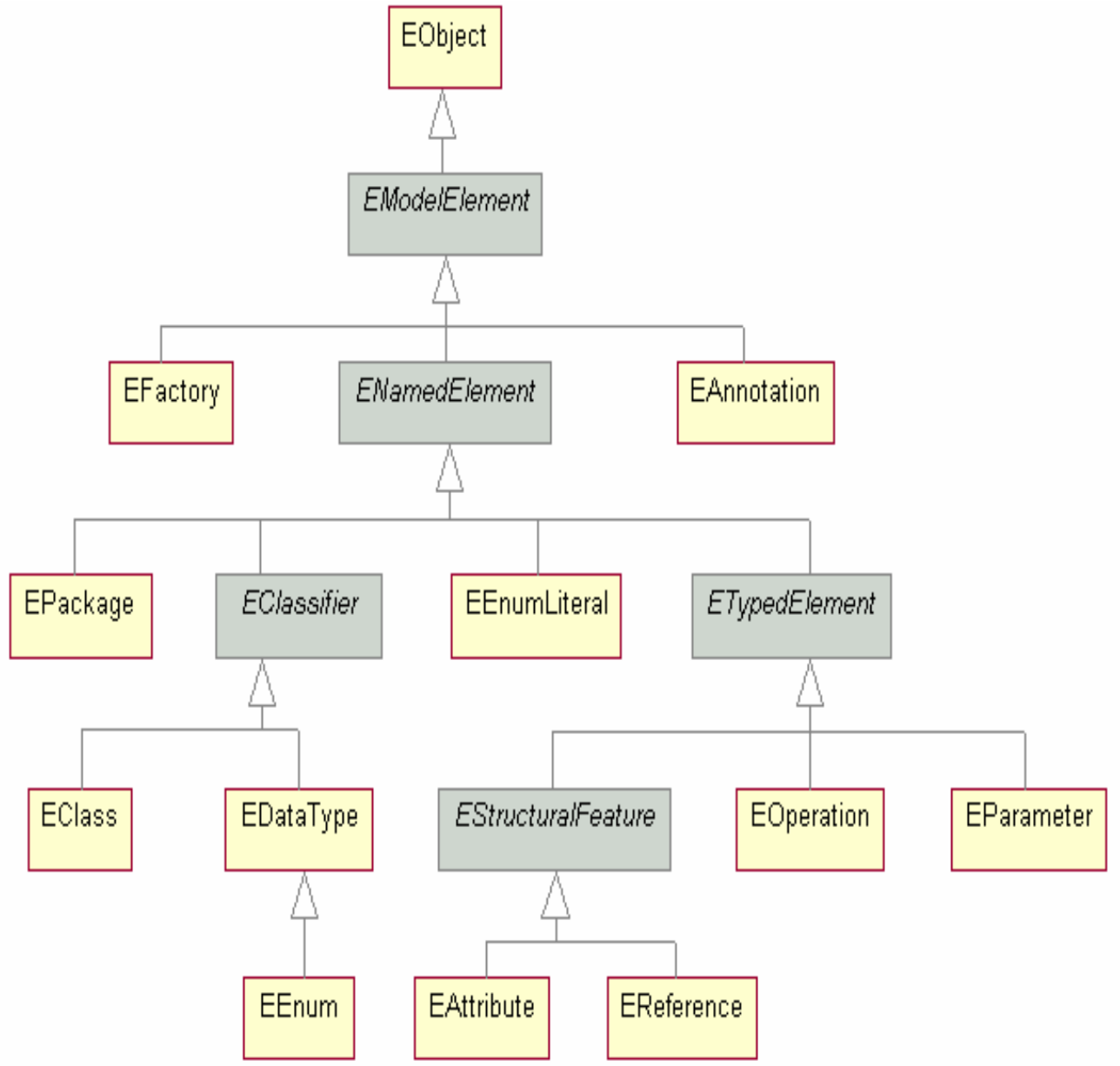


Figure 8: EMF Inheritance Diagram

7.1 ECLIPSE USE CASES

By analyzing the EMF model and the documentation available on its home page (www.eclipse.org), some of the features supported by EMF can be mapped to the library of few specific design patterns. The following table illustrates the four patterns mined in the EMF model:

Problem	Solution	Pattern Used
Change in objects should be notified to related objects in EMF model	EMF provides a notification mechanism inherently available in all the sub-classes of EObject	Observer Design Pattern
Objects should be able to dynamically communicate and share data with each other	Dynamic objects are created along-with notifications, which provide interfaces to objects	Adapter Design Pattern
Objects should be invoked or loaded only when they are actually needed in the system.	EMF stores objects as resources that can be invoked using proxies.	Proxy Design Pattern
Dynamic class instantiation and object creation is required in EMF to provide interfaces to mismatching objects.	EMF uses factory method to instantiate adaptor classes that dynamically support objects with different types.	Factory Method Design Pattern

Table 3: patterns mined in the EMF model

7.2 USE CASE DIAGRAM

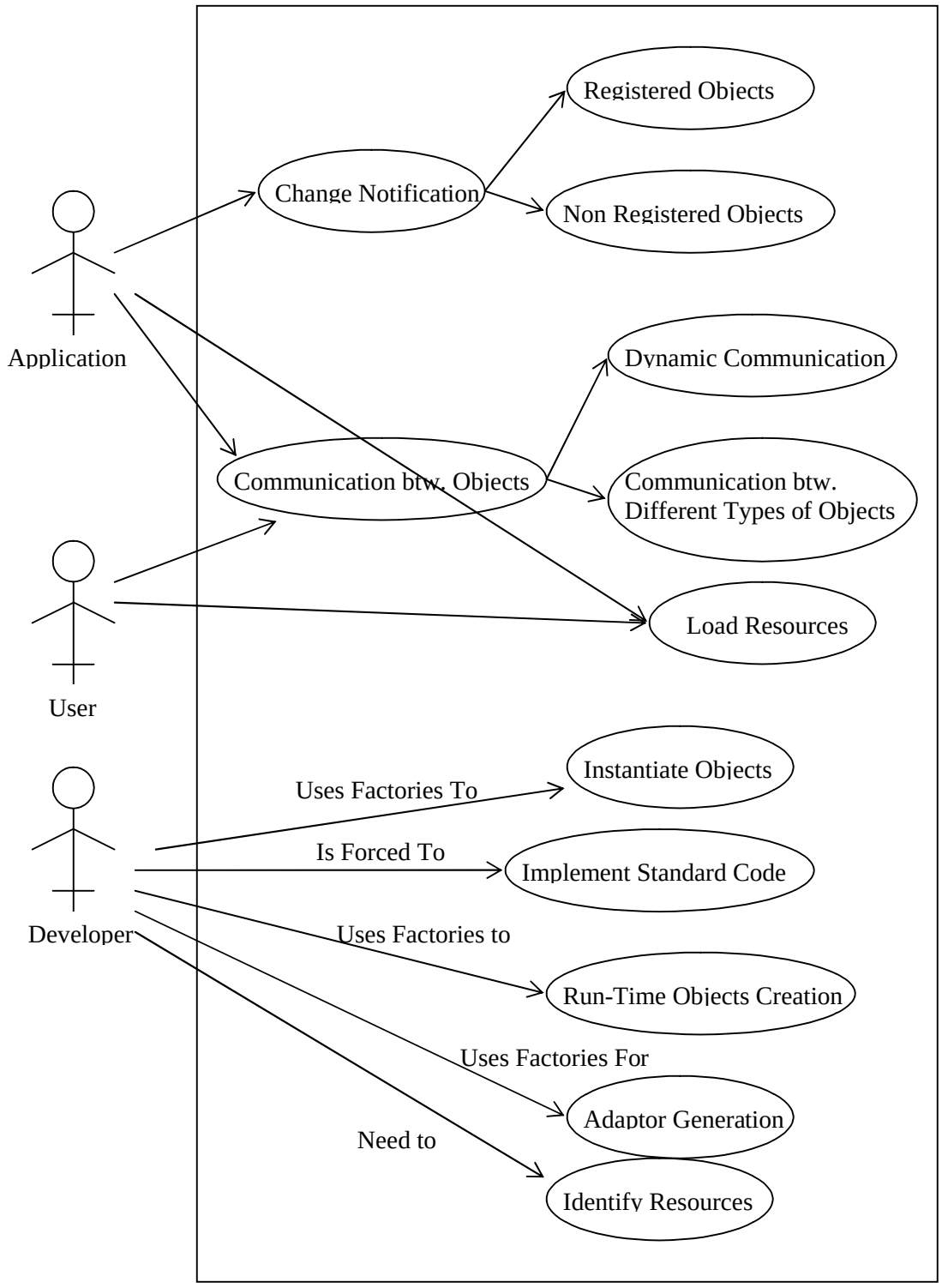


Figure 9: Use case Diagram

7.3 OBSERVER

a. Definition

Defines dependency between objects so that when one object changes state, all it dependents are notified and updated [4].

b. Problem Statement:

EMF supports persistence and dynamic invocation of objects instantiated from its ECore meta-data structure. The implementation of these objects can produce dependencies among objects, which requires implementation of a notification mechanism, so that when one object changes state all its dependents are notified. However, there is a need for efficient implementation of this functionality, so that the performance of objects is not affected that do not need notification functionality.

c. Use Cases

Use Case ID: 001

Pre-Req. Use Cases: Null

Use Case Purpose: Objects in EMF should provide built-in support for change notification so, that when an object changes its state, all the objects related or dependent to this object are notified.

Actors:
User: Invokes operation on an object
Application: Stores or updates an XML object

Use Case: Object Change Notification

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
7. A number of objects are instantiated by application and stored as XML schemas.
8. User requests a service which invokes operation on one of the objects.
9. The object contains reference ids to all the objects related to it.
10. The invoked operation updated status of the object. For instance, some variable is initialized in the object.
11. The listeners implemented in EMF inform all associated objects about the state change of object.
12. All the associated objects stores the updated state of the object under consideration.
13. Updated object is again saved as XML object for new operations.

Termination Condition:
Change in object is successfully notified to all associated objects and updated object is successfully saved as XML object.

Use Case ID: 002

Pre-Req. Use Cases: 001

Use Case Purpose: Objects should allow registration of listeners so that only registered listeners are notified about state change.

Actors:
User: Invokes operation on an object
Application: Stores or updates an XML object

Use Case: Object Change Notification

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
Application objects called 'Notifiers' keep references to the objects that need to update themselves and objects that want to publish there state when a change occurs to them.
User requests a service which invokes operation on one of the objects.
The object contains reference ids to all the objects related to it.
The listeners implemented in EMF inform all associated objects about the state change of object.
All the objects that are not registered with 'Listener' are not affected by the change notification mechanism.

Termination Condition:
Change in object is successfully notified to only associated objects and um-registered objects are not affected.

Use Case ID: 003

Pre-Req. Use Cases: 001

Use Case Purpose: Objects change notification functionality should not invoke any function when no listeners are registered to the objects; this is to safe system resources and improves the performance.

Actors:
User: Invokes operation on an object
Application: Stores or updates an XML object

Use Case: Object Change Notification

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which invokes operation on one of the objects.
The object contains reference ids to all the objects related to it however it does not need to update other objects about its state change.
On the front screen, let say a drawing object is moved to a different location.
No object is updated about the change in location as it is private to an object to store its position.

Termination Condition:
Drawing object is successfully moved to new location without affecting the objects that it uses for normal operations.

Use Case ID: 004

Pre-Req. Use Cases: 001

Use Case Purpose: Standard implementation support offered by EMF encodes notification methods in set() methods i.e. whenever an object is set to a new state, the corresponding objects are notified.

Actors:
User: Invokes operation on an object
Application: Stores or updates an XML object

Use Case: Object Change Notification

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which invokes operation on one of the objects.
User performs an operation on the object i.e. changes the value of the object's elements through set method.
The set method inherently calls all other objects that provide reference to this object.
All referenced objects are informed about the latest change in the state of the object.

Termination Condition:
All referenced objects are successfully notified about the state change of the object.

d. Observer Structure in EMF

Figure 10 illustrates a simple structure of the use of observer pattern in EMF model. The objects in this case can be resources stored as XML or active objects running in the application. The change notification object works as listener to objects that associate with it. Whenever a state change occurs in an object, the notification object notifies all the related objects about the state change.

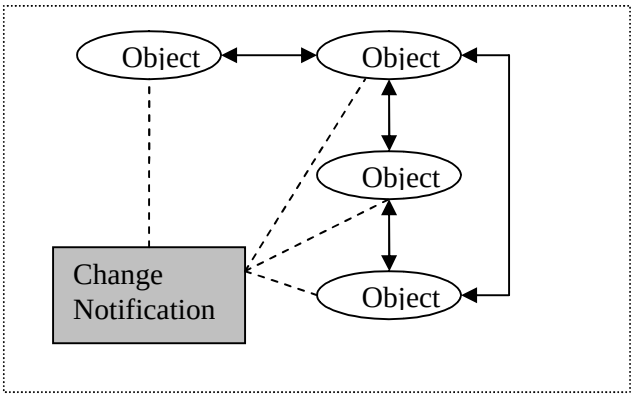


Figure 10: Change Notification Implementation in EMF

e. EMF Solution Implementation using Observer Pattern:

In the ECore model implementation, class EObject extends Notifier class, since EObject is a parent class in this model so notifier functionality is inherently accessible to all sub-classes in the model. Thus every generated EMF class is also a Notifier that can send notification whenever a change takes place. This is an important property of EMF model that allows EMF objects to be observed, to update views and to update dependent objects [6]. In essence, this notifier functionality in EMF implements *observer* design pattern in its structure.

The standard EMF implementation is based on most common function calls, objects interaction and notification implementation. Set and Get methods are among the default functions implemented in EMF framework. These methods can be further extended for application specific needs of the software. Default implementation of some part of setFunction() method is shown below. Whenever setFunction method is invoked, the function method sends notification to any observers that may be listening to the object. eNotificationRequired method is used to optimize the case if there are no observers used by the object, this method simply checks if there are any observers attached to the object [5]. In case, there is no observer attached to the object, this Boolean function simply returns a false value, improving system run-time performance.

```
Public interface EObject extends Notifier {
// Attributes
}

Public void setFunction() {
// set function implementation
If (eNotificationRequired())
    eNotify(new EnotificationImpl(this,
                                  Notification.SET,
                                  ... ));
.....
}
```

f. Consequences

EMF implements notification mechanism at the root level of model, this provides inherent availability of notification functions in objects created in the model. This strategy increases the maintainability and flexibility of the system, where any change in implementation can be introduced at abstract level of the system. Further it separates application logic from observers in the system. However, this can negatively affect the performance of the system. In case, when a large number of objects do not need to implement listeners in their functionality, extra handling of objects may be needed.

g. Variation to Observer Pattern

In its pure form the observers directly interact with their subjects, however in EMF observer work within adaptors of the system. All the notification communication between observers and the subjects is passed through adaptors. Thus an additional layer of adaptors is generated. The purpose of such an implementation in EMF is that observers are inherently available in all objects including adaptors. When adapters are created using adaptor factories, they work as encapsulation to observer functionality as well.

7.4 ADAPTOR

a. Definition

Adaptor provides clients expected interfaces between classes so that classes can work together [4].

b. Problem:

Objects in EMF need to dynamically communicate to use or provide service to the other objects.

c. Use Cases

Use Case ID: 005

Pre-Req. Use Cases: null

Use Case Purpose: Dynamic communication between objects in EMF should be supported.

Actors:
User: Invokes operation on an object which dynamically communicate with other objects
Application: Stores or updates an XML object, created interfaces for communication

Use Case: Dynamic Communication between objects

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which invokes operation on one of the objects.
User performs an operation on the object, which needs to interact with other objects to complete its operation.
Application creates an adaptor object that works as an interface between invoker and receiver objects.
The invoker object sends its request to adaptor object.
Adaptor receives the request, locates the receiver and sends it to receiver for processing.
Same process is followed when response is returned from receiver.

Termination Condition:
Invoker object has successfully received the requested information.

Use Case ID: 006

Pre-Req. Use Cases: 005

Use Case Purpose: Objects with different types should be able to communicate and transmit data in EMF.

Actors:
User: Invokes operation on an object which dynamically communicate with other objects
Application: Stores or updates an XML object, created interfaces for communication

Use Case: Dynamic Communication between objects

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which invokes operation on one of the objects.
User performs an operation on the object, which needs to interact with other objects to complete its operation.
Application creates an adaptor object that works as an interface between invoker and receiver objects.
The invoker object sends its request to adaptor object.
Adaptor receives the request, locates the receiver, converts the input information compatible to the receiver object and sends it to receiver for processing.
Same process is followed when response is returned from receiver.

Termination Condition:
Invoker object has successfully received the requested information.

Use Case ID: 007

Pre-Req. Use Cases: 005

Use Case Purpose: Factories should be used to generate objects at run-time in EMF.

Actors:
User: Invokes operation on an object which dynamically communicate with other objects
Application: Stores or updates an XML object, created interfaces for communication

Use Case: Dynamic Creation of objects through factories

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which invokes operation on one of the objects.
User performs an operation on the object, which needs to interact with other objects to complete its operation.
Application creates an adaptor object that works as an interface between invoker and receiver objects.
Application invokes the factory method operation, which at run-time instantiates an adaptor object which is capable to communicate between both the sender and receiver objects.
Invoker object sends its request to adaptor object.
Adaptor receives the request, locates the receiver, converts the input information compatible to the receiver object and sends it to receiver for processing.
Same process is followed when response is returned from receiver.

Termination Condition:
Invoker object has successfully received the requested information.

d. Adaptor Structure in EMF

Factories in EMF are used to instantiate different types of objects; one purpose of these factories is to produce adaptors that can work as interface between different types of objects. A simple structure of adaptors generated for objects of EMF model is shown in figure 11.

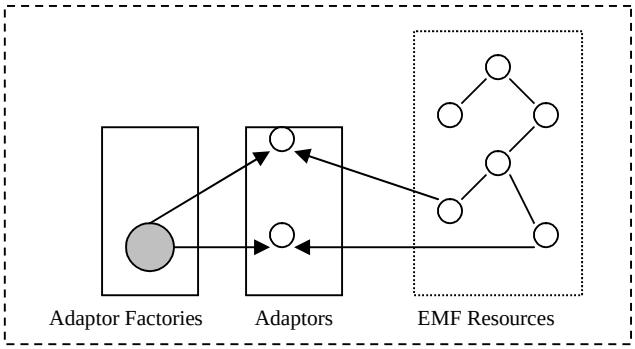


Figure 11: Adaptor Creation for EMF Resources

e. EMF Solution Implementation using Adaptor Pattern

From a model specification, EMF produces a set of Java Classes to represent the model, as well as a set of adapter classes that enable viewing and editing. The observers and adapters are used in collaboration in EMF model. As they both are inherited from same base class, so there presence in every EMF model child class can be used by first adding adopter to a particular object and then invoking notification function to inform other objects about the state change. Using adaptor as an observer, it can be attached to any EObject as follows [5]:

```
Adaptor poObserver = ….
aPurchaseOrder.eAdapters().add(poObserver);
```

The extension to the behavior of observers used in EMF model is done using an adaptor factory [5]. An adaptor factory is used to adapt an object with the required extension as illustrated in following code:

```
PurchaseOrder aPurchaseOrder = ...
AdapterFactory somePOAdapterFactory = ...
Object poExtensionType = ...
if  (somePOAdapterFactory.isFactoryForType(poExtensionType))
{
Adapter poAdapter =
somePOAdapterFactory.adapt(aPurchaseOrder, poExtensionType);
...
}
```

f. Consequences

The generic support to generate a variety of adaptors in EMF affects quality attributes both positively and negatively. It increases the flexibility and maintainability of the system by allowing to generically offering the support for new data and objects types. Even the objects with different types can flexibly communicate with adaptor support

offered by EMF. However, EMF generates adaptor for all kinds of objects, including serialized objects in the system which can have a negative affect on the performance of the system.

c. Variation to Adaptor Pattern

Adaptor pattern is implemented in its pure form in EMF.

7.5 FACTORY METHOD

a. Definition

Defines interface for creating an object and leaves instantiation decisions to sub-classes [4].

b. Problem

EMF uses adaptors that support different types of objects, so how should these adapters be generated in the system to support diversity in the target objects?

c. Use Cases

Use Case ID: 008

Pre-Req. Use Cases: null

Use Case Purpose: Classes in the ECore model should be instantiated by the use of factory methods. This is to force the abstraction to the core model and extension in the application model.

Actors:
User: Invokes operation on an object which dynamically communicate with other objects
Application: Stores or updates an XML object, created interfaces for communication

Use Case: Creation of EMF objects through factories

Invocation Conditions: All the object instantiation calls are made to respective factory methods.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
User requests a service, which invokes operation to create a new object.
The operation invokes operation on factory method class available in EMF object.
Factory class has 'instance creation' link with packages available in EMF model.
In accordance with the request, an object of requested type is created at run-time by factory methods.
The object is returned as a parameter to the invoker function.

Termination Condition:
Invoker object has successfully received the requested object.

d. Factory Method Structure in EMF

Similar to java structure, the ECore model provides packaged support to group related classes and factories are used to instantiate the instances of the classes in the package. From the inheritance diagram of figure 3, we capture the following core diagram to illustrate the implementation of factory method in ECore.

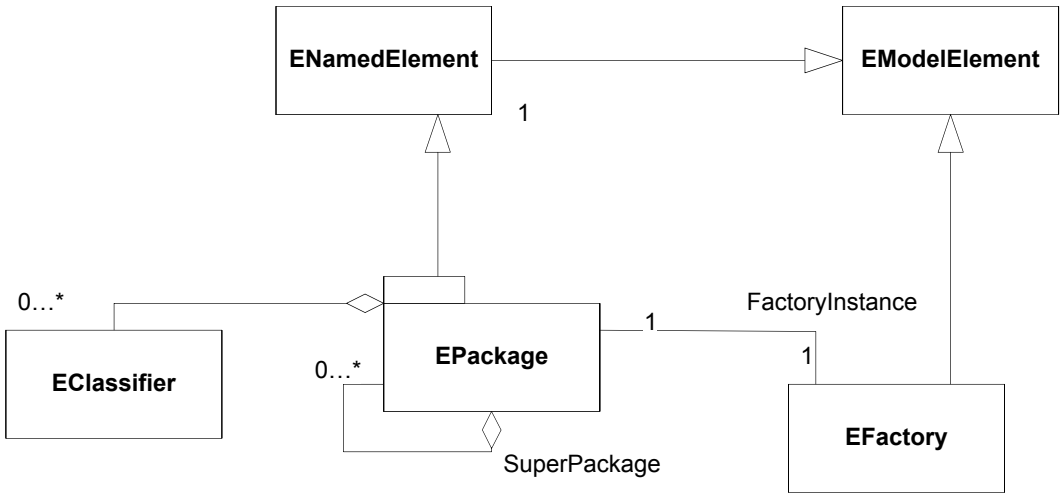


Figure 12: Factory Method Structure in EMF

e. Implementation

Since factories in EMF model do not have further instance data, they need not be explicitly modeled; rather they are created on-demand in the structure [6]. As the factories do not inherit any structural features so there role is more behavioral in nature. As shown in the figure in previous section, a factory supports methods of create, createFromString and convertToStrings. These methods are explicitly used to instantiate classes and support different data types. For instance to support adapters with a different data type target class, these methods are used to convert data type values to and from strings. The methods createFromString and convertToStrings can further be manually updated to support additional data types and objects. Complete list of data and object types dynamically supported by EMF factories can be found in [6]. In fact, EMF provides a variety of factories that serve different purposes in implementation; it enforces implementation rules to generate a variety of objects using factories. For instance, factories are used to serialize objects, generate itemProviders, create adaptors etc. In this work we have specifically focused on the factory method support for adaptors using factory method pattern.

f. Consequences

Use of factory method pattern in EMF significantly improves the maintainability, re-usability and flexibility of the system by separating abstract behavior from concrete implementation.

g. Variation to Factory Method Pattern

Factory method pattern is implemented in its pure form in EMF.

7.6 PROXY

a. Definition

Provides control access mechanism on objects so that when requested, there services are easily accessible [4].

b. Problem

EMF provides support to save resources as objects, which need to be uniquely identified and robustly accessible when their services are requested.

c. Use Cases

EMF framework uses proxy design pattern to perform following functions:

Use Case ID: 009

Pre-Req. Use Cases: null

Use Case Purpose: Resources are used as persisted documents, loaded and saved as files, and need to be referenced.

Actors:
Application: Stores or updates an XML object, created interfaces for communication, provides reference to stored objects

Use Case: Proxies are used to load resources in EMF projects

Invocation Conditions: Application has successfully created objects that can be invoked, stored or updated in the system.

Flow Conditions:
1. Application objects are created from EMF model.

Flow of Events:
A number of objects are instantiated by application and stored as XML schemas.
User requests a service which references to many of the stored objects.
Instead of loading all referenced objects at once in memory, the system loads only the objects that are directly involved in current operation.
The system uses proxies to mirror all other referenced objects as available in memory.
When a request is generated to perform operation on any such referenced object, proxy is used to load only the requested object that has direct link to the invoker.

Termination Condition:
Proxy successfully provides a mirror to stored object as these are available in memory and loads only the objects that are directly involved in computation.

d. Proxy Structure in EMF

The link between resources in the same resource set in EMF is implemented using proxies. EMF uses these proxies to support on-demand loading of referenced resources. These resources in actual contain objects. The XML support offered by EMF is used to store model and objects data and each resource in EMF is identified by a unique mark called unified resource identifier (URI) in EMF [3]. It is possible to provide one mark to more than one resources of same category; often called a resource set.

A general structure of resources persisted in EMF using XML is shown in figure 13, with some source code implementation. The createResource function contains the actual resource to be invoked while next tag refers to the related resources.

Figure 13: Resource Persistence in XML format [5]

```
poResource =
     ...createResource(..."p1.xml"...);
poResource.getContents.add(p1);
poResource.save(...);

<PurchaseOrder>
     <shipTo>John Doe</shipTo>
     <next>p2.xml#p2</next>
</PurchaseOrder>
```

e. Implementation

The actual implementation of these proxies in EMF can be illustrated by figure 7. In EMF structure, class EReference has association relationship to EClass, the class EReference defines attribute of resolveProxies that is used to persist objects. Further these proxies are used to reference objects that are loaded in memory. In fact the automatic implementation of proxy is performed only for objects that contain boolean value of resolveProxies as true. If an object is accessed in EMF, and its resource is already persisted, then the object is returned and only the resource is loaded again [6].

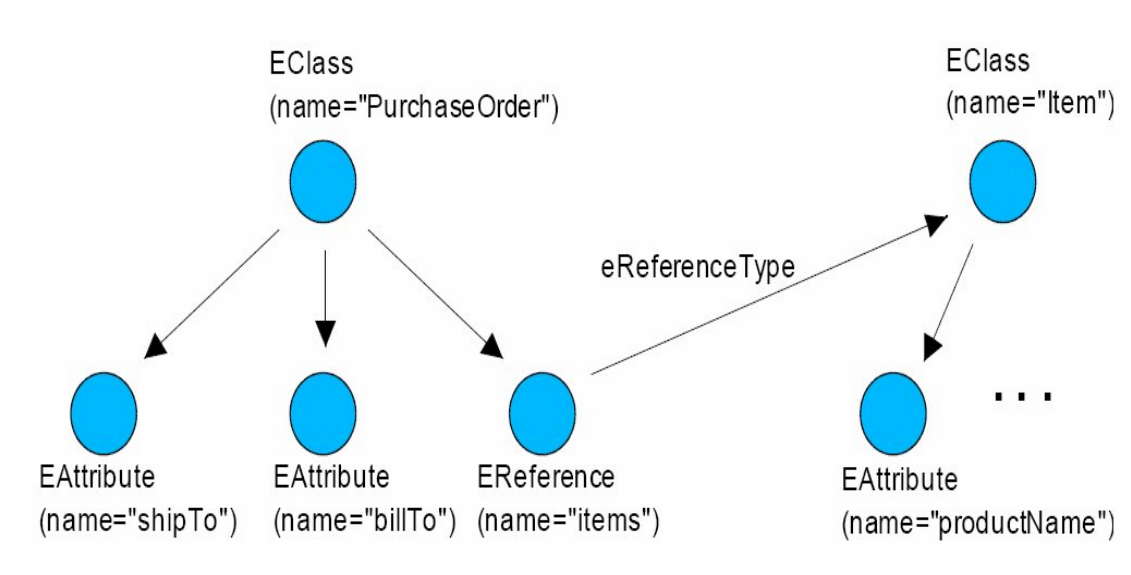


Figure 14: Reference creation between objects [5]

Sample XML code to establish references shown in figure 14 is listed below.

```
<eClassifiers xsi:type="ecore:EClass"
                  name="PurchaseOrder">

<eReferences name="items" eType="#//Item"
     upperBound="-1" containment="true"/>

<eAttributes name="shipTo"
              eType="ecore:EDataType
              http:...Ecore#//EString"/>

<eAttributes name="billTo"
               eType="ecore:EDataType
             http:...Ecore#//EString"/>
</eClassifiers>
```

f. Consequences

Proxy brings additional implementation overhead in the system which can increase the complexity of the system, decreases maintainability and affects performance of the system.

g. Variation to Proxy Pattern

Proxy pattern is implemented in its pure form in EMF.

7.7 INCORPORATING NEW PATTERNS IN EMF

a. Definition: Design Pattern Singleton

Ensures that a class has only one instance, and a global point of access is used to provide or request the services of associated classes [4].

b. New Use Case Description as Requirement

To optimize the system performance and to avoid redundant objects, it is appealing to have one adapter for one or more resources of same type that are responsible to handle all requests made to the object.

c. Structure

Introduction of Singleton design pattern does not affect the basic structure of EMF. Similar to the adaptor and notification patterns discussed in previous sections, the singleton pattern is also extended by the base class EObject. A new interface class Singleton is added in EMF model which extends to the EObject class. Thus all objects generated in EMF model work as a singleton. Figure 9 shows the high level structure of adaptor implementation using singleton pattern.

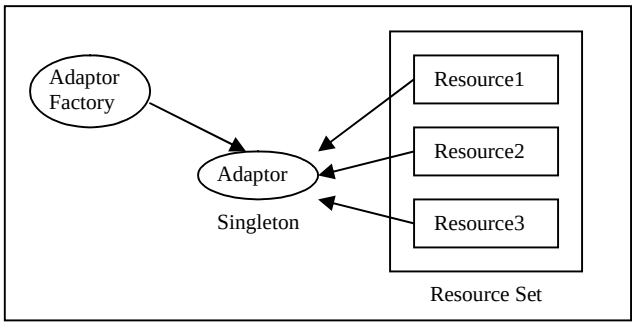


Figure 15: Adaptor implementation as a singleton

d. Collaborations

In the EMF structure, class EObject extends Singleton interface.

e. Implementation

As EObject is the parent class in the structure so all objects generated in EMF structure implement singleton functionality. A pointer to singleton is inherited to every adaptor generated from adaptor factory. Once an instance to a class that inherits Singleton is created, the static pointer instance is initialized and remains alive unless the invoker destroys it. When an attempt is made to generate a new instance of the class, the function first checks that pointer does not already contain an instantiated value. If the pointer is already initialized in the system, the same adaptor is passed to the requesting function, otherwise adapter with new initialized pointer value is returned.
Adaptors in EMF are responsible to handle the behavior extension in objects and to receive notification alerts; we use same adaptors to implement the functionality of

singleton. An enhanced version of adaptor class implementation in EMF will look like this:

Package
org.eclipse.emf.common.notify

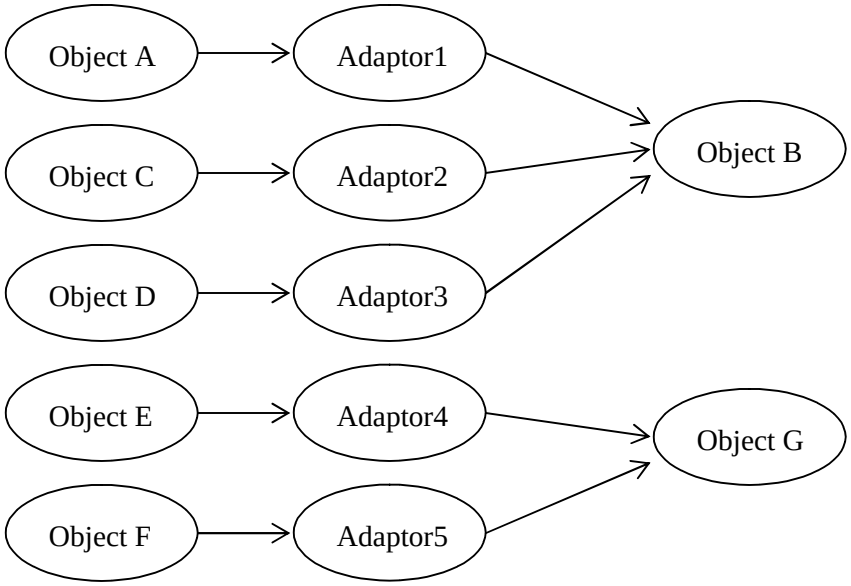
Adaptor Class Interface
Public interface Adaptor {
static Singleton* initializeInstance();
// attributes
};

f. Consequences

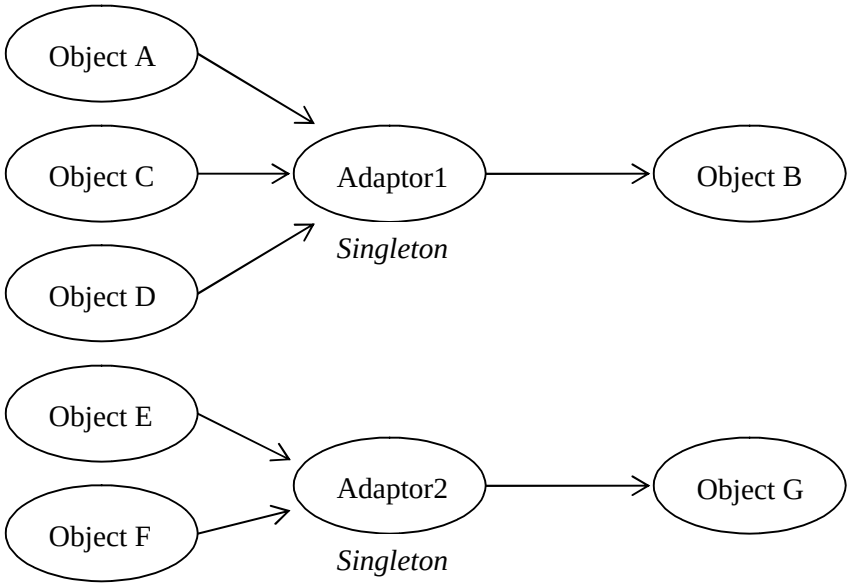
Implementation of Singleton pattern can affect the complexity and reliability of the system due tot the presence of a single point of access to a number of objects, however it provides efficient use of resources, reduces redundancy and saves memory of the system.

One important aspect of mining these patterns is that they address specific quality attributes of the system and listing them in order helps architect to relate them to the overall quality of the system. For instance, one benefit of using Proxy design pattern is that it helps to save the upload time of saved resources in memory. Any new implementation on Eclipse platform can benefit from this structure to use Proxy objects to upload resources. By specifically focusing on the qualities inherent in design and by relating those to patterns help architect to address the weaker areas of application.

During our work we identified the problem of *multiple* Adaptor objects created in EMF system (as discussed in section 7.5). For instance, object A communicates with object B using Adaptor1 while object C also need a new adaptor(Adaptor2) to communicate with objectB as shown in the figure. For each new connection to communicate with objects with different types, a new instance of adaptor object is generated. This can cause an excessive uses of memory and resources of the system which can affect the performance of the EMF system and the systems built on EMF negatively.



To solve the above mentioned problem, which we came across while the discovery of Adaptor pattern. We decided to introduce a new pattern in EMF structure, namely Singleton pattern, which restricts creation of multiple instances of same object. As shown in the figure below this new pattern allows to initialize Adaptor pattern only once. The detail implementation of this new pattern is explained in section 7.6. We expect that this reasoning methodology will not only help in improvement of the quality of architecture but documenting the patterns used to design software architecture will improve the readability and understandability of the architects about the system.



8 CONCLUSION

This paper discusses the use of software patterns in the design of software systems, consequences of these patterns on the quality attributes of the system and using use case to mine patterns in existing software systems. With the help of use cases, we mine four patterns in the Eclipse (EMF), and show their implementation structure in the system. Since EMF is a base platform used to design new models and meta-models so, the qualities inherited in the EMF model directly influence the quality of models generated from its structure. This scheme provides a systematic way to the engineers by looking in use cases and relating these use cases to patterns modeled in the software system. Further, we illustrate with the addition of a specific design pattern in EMF structure to illustrate the enhancement in the quality of the EMF model. Surfing the documentation and information related to Eclipse proved helpful in identifying use cases in the system and hence eventually leading to the mining of patterns. This explicit representation of patterns mined in the Eclipse system helps users to reason about the quality attributes and in the better understanding of the system.

We expect that the study performed in this paper will help readers to utilize use cases to realize the essence of software patterns in the system. This will help in considerable reduction of time and effort spent to discover patterns, highlighting the concept of relating patterns to use cases, and to reason about the quality of the system.

9 REFERENCES

Literature

[1] Douglas C. Schmidt, Applying Design Patterns to Flexibly Configure Network Services in Distributed Systems, Department of Electrical and Computer Science, University of California, Irvine 92607, May 6-8, 1996

[2] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects, Volume 2, Willey Series in Software Design Patterns, John Willey & Sons Ltd., Copyright 2000

[3] D. Leroux, M. Nally and K. Hussey, Rational Software Architect: A Tool for Domain Specific Modeling, IBM Systems Journal, Volume 45, No. 3, 2006

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns – Elements of Reusable Object Oriented Software*, Addison Wesley Professional Computing Series, Copyright 1994

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, Pttern – Oriented Software Architecture, A System of Patterns. John Wiley and Sons, West Sussex, PO19 1UD, England, 1996

[6] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick and Timothy J. Grose, Eclipse Modeling Framework, The Eclipse Series, Addison Wesley, Copyright 2004

[7] Grady Booch, James Rumbaugh and Ivar Jacobson, 'Unified modeling language user guide', Addison Wesley, 1st Edition, October 20, 1998

[8] Victor F. A. Santander and Jaelson F. B. Castro, *'Deriving Use Cases from Organizational Modeling'*, Universidade Federal de Pernambuco – Centro de Informática Cx. Postal 7851, 2002, BRAZIL

[9] Karl E. Wigers, 'Requirements Engineering', 2nd Edition, Microsoft Press, copyright 2003, ISBN: 0735618798.

[10] Björn Regnell, *'Hierarchical use case modeling for requirements engineering'*, Department of Communication System, Lund Institute of Technology, Lund University, 6th September, 1996

[11] J. Dorr, D. Kerkow, A. Von Knethen and B. Paech, 'Eliciting Efficiency Requirements with Use Cases', Fraunhofer IESE, Kaiserslauten, Gernmany

[12] Eman Nasr, John McDermid and Guillem bernat, 'Eliciting and Specifying Requirements with Use Cases for Embedded Systems', Department of Computer Science, University of York, UK, 2002

Web-Resources

[13] Alistair Cockburn , *'Structuring Use Cases with Goals'*, Technical Report January 1995, http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm,

[14] Elena Litani, Ed Merks and Dave Steinberg, *'Discover the Eclipse Modeling Framework (EMF) and Its Dynamic Capabilities'*, 26th August, 2005, http://www.devx.com/Java/Article/29093/0/page/1

[15] *'Use Cases'*, TechTarget, Copyright 1999 - 2006, http://searchwin2000.techtarget.com/sDefinition/0,,sid1_gci334062,00.html

[16] Software Engineering Institute, 4500 Fifth Avenue, Pittsburgh, PA 15213-2612, Copyright 2007, http://www.sei.cmu.edu