Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Mohamed Khider - BISKRA
Faculty of Exact Sciences, Natural Sciences and Life
Computer Science Department

Order Number:

Memoir

Presented to obtain the diploma of academic Master in

Computer Science

Option: **Software Engineering and Distributed Systems**

Software Architecture-Based Evolution for Component/Service-Oriented Systems

---

By:
Amani KHADRAOUI

Defended the ..../..../2020, in front of the jury composed of:

.....	...	President
Dr. Mohamed Lamine KERDOUDI	MCB	Supervisor
.....	...	Examiner

College Year: 2019 - 2020

République Algérienne Démocratique
et Populaire
Ministère de L'Enseignement Supérieur et de La Recherche Scientifique
Université Mohamed Khider - BISKRA
Faculté des Sciences Exactes, des Sciences de La Nature et de La Vie
Département d'Informatique

numéro d'Ordre:

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Parcours: **Génie Logiciel et Systèmes Distribuées**

Évolution à Base d'Architecture pour Les Systèmes Orientés Composants/Services

---

Par:
Amani KHADRAOUI

Soutenu le ..../..../2020, devant le jury composé de:

.....	...	Président
Dr. Mohamed Lamine KERDOUDI	MCB	Rapporteur
.....	...	Examinateur

Année Universitaire: 2019 - 2020

Acknowledgements

Before all, I thank **Allah** who gave me all the courage and the will to go till the end for the accomplishment of this work.

I express my deep gratitude and sincere appreciation to my teachers and supervisors **Dr. Mohamed Lamine KERDOUDI** for the time he devoted to me and for his follow-up during the period of realization of my project.

Also, I appreciate the support of **Ph.D. student Ikram MAOUCHE** for her time, assistance, and guidance to realize this work.

My thanks are also extended to the members of the jury for honoring me with their reading and their evaluation of my dissertation.

My finally thanks go to all my teachers at the Computer Science Department of Biskra.

Dedication

I dedicate this work to my dear parents, my father **Said** and my mother **Nabila** for their endless advice, encouragement, and support as a testimony of my gratitude, in the hope that they will be proud.

To my dear brother **Dr.Mohamed Amine**.
To my lovely sisters **Sara**, **Soumaya**, **Yousra**, **Sabrine**, and **Meriem Sirine**.

To my great professor, **Dr.Nesrine OUANNES** may God have mercy on her.
To all my family and friends.
To all my dear ones.

Abstract

Software architecture is a high-level abstract description of the system and its components. It plays an important role during the maintenance and evolution of the system, which saves time, money and effort. The evolution of the software architecture can be at design time (static evolution), in which there is no direct update or modification of the system and its components, or can be at the time of system execution (dynamic evolution), in which one can update and change both the architecture and the system while removing / adding components during the execution of this system. This work aims to help developers evolve their systems at runtime based on its dynamic software architecture. We have proposed a process, it is applied on a component-based application. The process first begins to recover the component-based architecture of the system at runtime. The second step is to evolve the system by manipulating this architecture. Changes to the architecture are reflected in the system.

Keywords: Software architecture, Software architecture evolution, Dynamic evolution, Component-based architecture, OSGi.

Résumé

L'architecture logicielle est une description abstraite de haut niveau du système et de ses composants. Elle joue un rôle important lors de la maintenance et de l'évolution du système, ce qui permet d'économiser du temps, de l'argent et des efforts. L'évolution de l'architecture logicielle peut être au moment de la conception (évolution statique), dans laquelle il n'y a pas de mise à jour ou de modification direct du système et et de ses composants, ou peut être au moment de l'exécution du système (évolution dynamique), dans lequel on peut mettre à jour et changer l'architecture et le système à la fois tout en supprimant/ajoutant des composants durant l'exécution de ce système. Ce travail vise à aider les développeurs à évoluer leurs systèmes à l'exécution en se basont sur son architecture logicielle dynamique. Nous avons proposé un processus, il est appliqué sur une application basée composants. Le processus commence premièrement à récupérer l'architecture basée composants du système au moment de son exécution. La deuxième etape consiste à evoluer le système en manipulant cette architecture. Les changements sur l'architecture est refléter sur le système.

Mots clés: Architecture logicielle, évolution de l'architecture logicielle, évolution dynamique, architecture basée sur les composants, OSGi.

Software Architecture-Based Evolution for Component/Service-Oriented Systems

Amani KHADRAOUI

September 15, 2020

Contents

List of Figures

General Introduction

Today, software architecture plays an important role during software systems maintenance and the evolution process. It gives an abstract description of its structure with system components information. It has the potential to provide a basis for managing software evolution, as there are interconnections between the described purposes of software engineering and software evolution [BCL12]. Architecture evolution can occur at the specification (design) time that is called a static evolution or during the execution of the system (at runtime) that is called dynamic evolution. If there are no changes to the system architecture at runtime that the software system architecture is a static architecture, this means there are no new connections between the system components, and the existing components are can not destroy, but a dynamic architecture if the system's components and connections can be created and destroyed at runtime according to the rules from design-time [Ryc17]. Software architecture evolution at runtime may ultimately lead to incorrect architectural configurations that can lead to major architectural violations at runtime [BR00]. In this work, we are interested to answer to the question : *How can we use dynamic software architectures to help developers for evolving and maintaining their systems at runtime?*

Based on the previous points, our main objective in this project is to propose a software architecture based evolution approach for assisting developers to evolve their systems at runtime. Our process is divided into two main sub-processes. The first one titled "Software Architecture Recovering", its objective recovering component-based architecture from applications at runtime. The second sub-process titled "Dynamic Software Architecture Evolution", its objective makes the recovered architectures interactive which allows developers to remove or update existing components (change state, modify names...), add new components, services, interfaces, or connectors at runtime. We apply our approach on the OSGi based applications. Indeed, we extended

an existing OSGi meta-model [Ker+19] to represent the OSGi software architectures that are recovered at runtime. Where, the existing meta-model allows to represents the systems only at design time. Finally, we developed a set of tools that offer: i) Recovering software architecture at runtime. ii) Presenting the recovered software architecture at runtime as an XMI file. iii) A graphical editor for visualizing and analyzing the recovered software architecture at runtime. iv) Evolving the system via the software architecture at runtime: Updating components, services, interfaces, etc. v) Automatic software architecture generation (the new one).

This thesis is organized as follow:

- **General Introduction:** the presents the context, the problem, and the aim of this work.
- **Chapter 1:** it presents the concepts and terminology of software architecture, their different styles of systems presentation, and their importance. Next, we present the concept of software evolution, and software architecture evolution.
- **Chapter 2:** introduces the software component and component model concepts. It presents also an overview of the different technologies and components models. Then, we introduce the Open Service Gateway Initiative technology (OSGi), one of the component models.
- **Chapter 3:** presents the state of the art of software architecture evolution. We present the existing techniques and works of software architecture evolution.
- **Chapter 4:** It presents our proposed approach for software architecture based evolution for component/service-oriented systems. We start first with an overview of the proposed approach. After that explain each step in detail. We present also in this chapter our proposed OSGi meta-model with an explanation of all its elements.
- **Chapter 5:** It introduces the global architecture of our tool *ArchDynEvol*, the details of the implementation of our proposed approach, the frameworks and tools that we have used. Finally, we provide an illustration of our tool via a concrete example.
- General Conclusion: we conclude and we introduce some perspectives, and future directions.

Chapter 1

Software Architecture and Software Architecture Evolution

1.1 Introduction

Software architectures of large systems are important for understanding the structure and behavior of a given systems during maintenance by offering an abstract representation of systems, and are important to the successful development of a software systems. In this chapter, we present the concepts and terminology of software architecture, its different styles of systems presentation, and its importance. Then, we present the software evolution concept, Lehman's laws, show an overview of its process, and its importance. Finally, we introduce the software architecture evolution concept and how can be affected by architectural dimensions, and its process.

1.2 Software Architecture

In this section, we present the different definitions of software architecture, its most famous styles and its importance.

1.2.1 Software Architecture Definition

There are a different definition of software architecture, most famous and accepted one is proposed by Rick Kazman et al. [BCK03]: *"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally*

visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural." From this definition:

- Software architecture is a set of structures. Structure is a set of components or elements keep together by a relation. Structure has three categories (Module, Component-And-Connector, Allocation) and this different structures are important in the design, documentation and analysis of architectures.
- Software architecture is an abstract representation with more details about components and how are used, related and interacted.
- Every system has a software architecture.

In summary, software architecture gives an abstract description of its structure. This structure gives information about components of system. Software architecture plays an important role in different aspects of software development and maintenance.

1.2.2 Software Architectures Styles

Software architecture styles are an abstract framework developed for a family of systems to provide solutions to problems that arise in the software life cycle process. There are set of principles and guidance's to define the vocabulary of components and connectors of that style [Akm+17]. Existing a different styles of software architectures, we give here the most famous styles client-service architectures (CSA), object-oriented architectures (OOA), component-based architectures (CBA) and service-oriented architectures (SOA).

1.2.2.1 Client-Server Architecture Style

The client-server architecture style describes distributed systems that involve separate client and server system, and a connecting network [Con+09]. Clients send requests to Server and wait for a reply. A server receives a request from a client and sends it the reply. Figure 1.1 represent the relationship between server and client [OLB16].

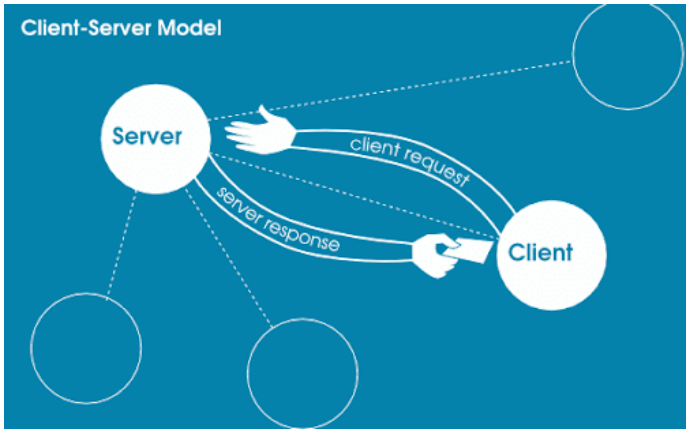

Figure 1.1: Client-Server Model.

Key Client-Server Architecture Principles

The main principles of client-server architecture style and how it works:

- The server is a service provider and the client is a consumer of services.
- Always the client who start the service request and the server passively waits for clients requests.
- Sharing resources such that, a server treats several clients at the same time and controls their access to resources.
- The data are managed centrally on the server. Clients remain individual and independent.

Benefits Of The Client-Server Architecture Style

The most important benefits of this architectural style are [Con+09]:

- **Higher Security:** All data is stored on the server that offers a greater control of security.
- **Centralized Data Access:** Data is on the server access and updates any data are easier to administer.
- **Ease of Maintenance:** There are roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade or relocation.

Limits Of The Client-Server Architecture Style

The client-server architecture style limits are:

- Overloaded servers and that when there are frequent simultaneous client requests, server severely get overloaded.
- Impact of centralized architecture and that when a critical server failed, client requests are not accomplished.

1.2.2.2 Object-Oriented Architecture Style

Object-oriented architecture is a design paradigm based on the division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object. Objects are discrete, independent, and loosely coupled; they communicate through interfaces, by calling methods or accessing properties in other objects, and by sending and receiving messages as Figure 1.2 shown.

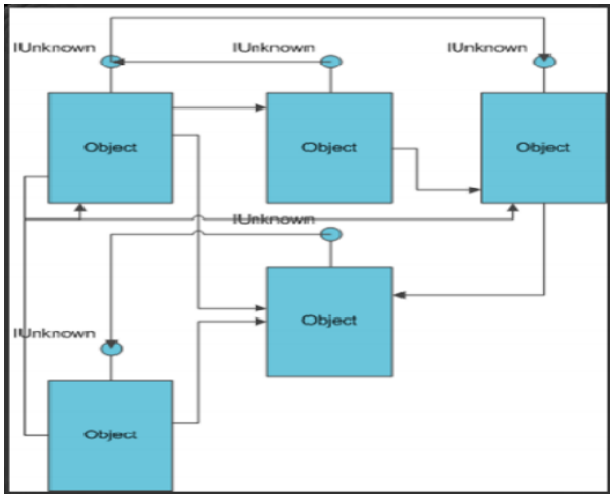

Figure 1.2: Object-Oriented Architecture objects interact [Akm+17].

Key Object-Oriented Architecture Principles

The key principles of the object-oriented architecture are:

- **Inheritance:** Where objects can inherit the characteristics of other objects.
- **Polymorphism:** Giving an object multiple forms.
- Abstraction, is process where only relevant data is presented without any details.

- **Encapsulation:** The internal of an object can be hidden from others so that only that object can manipulate its own state and variables.

Benefits Of Object-Oriented Architecture Style

The most important benefits of this architectural style are:

- **Understandable:** It maps the application more closely to the real world objects.
- **Reusable:** It provides for reusability through polymorphism and abstraction.
- **Testable:** It provides for improved testability through encapsulation.

Limits Of Object-Oriented Architecture Style

The object-oriented architecture style limits are [Akm+17]:

- Service integration and strong coupling between objects.
- Strong coupling between super classes and sub classes, swapping out super classes can break sub classes.

1.2.2.3 Component-Based Architecture Style

Component based architecture is an architecture that focuses on decomposing software design into functional or logical components with their own methods, events and properties [Akm+17]. It provides a high level of abstraction.

Key Component-Based Architecture Principles

The main principles of component-based architecture style are:

- **Extensible:** A components can be extended from existing components to provide new behavior.
- **Replaceable:** Components may be readily substituted with other similar component.
- **Encapsulated:** Components exposes interfaces and hides the details of the internal processes or any internal variables or state.
- **Independent:** Components can be designed to have minimal dependencies on other components.

- **Reusable:** Components are designed to be reused in different scenarios in different applications.
- **No Contexts Specific:** Components are designed to operate in different environment and contexts. Specific information such as state data should be passed to the component instead of being included in or accessed by the component.

Figure 1.3 abstract the major principles of component based architecture style.

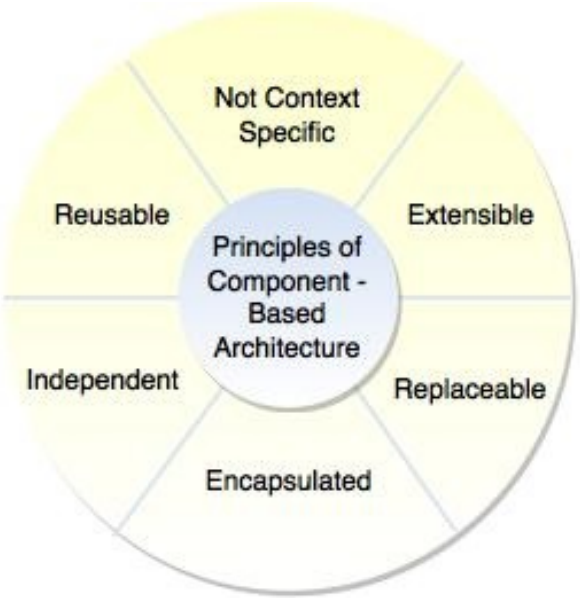

Figure 1.3: Principles of Component-Based Architecture.

Benefits Of Component-Based Architecture Style

The most important benefits of component-based architectural style are [Con+09]:

- **Reduced Cost:** The use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of Development:** Components implement well-known interfaces to provide defined functionality allowing development without impacting other parts of the system.
- **Ease of Maintaining/Update:** It is easy to maintain and update the implementation without affecting the rest of the system.

Limits Of Component-Based Architecture Style

The component-based architectural style limits are [Akm+17]:

- **Complexity:** While this style of architecture is designed to reduce complexity of systems it introduces a different type of complexity in terms of component-to-component interactions.
- **Testing:** Can be difficult if the component doesn't come with its own execution environment.

1.2.2.4 Service-Oriented Architecture Style

Service oriented architecture is an architecture style supports service orientation, and it makes application functionality to be provided as a set of services, and the creation of applications that make use of software services. Figure 1.4 represent the basic components in service oriented architecture, service provider, service users or consumer and service registry [Akm+17].

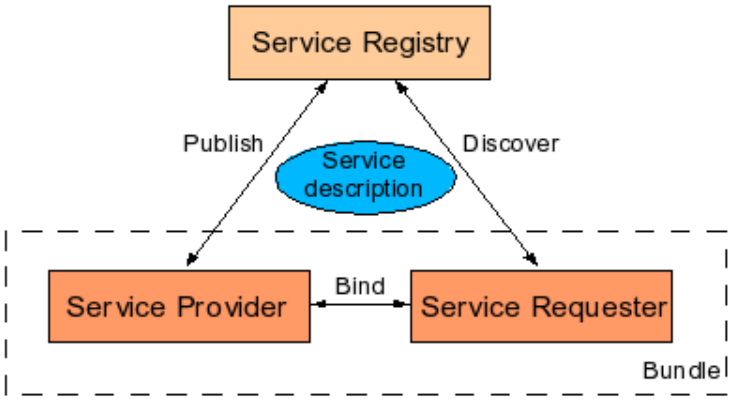

Figure 1.4: Service-Oriented Architecture.

Key Service-Oriented Architecture Principles

The main principles of this architectural style are [Con+09]:

- **Autonomous Services:** Each service is maintained, developed, deployed and versioned independently.
- **Distributable Services:** Services can be located anywhere on a network, locally or remotely.

- **Loosely Coupled Services:** Each service is independent of others, and can be replaced or updated without breaking applications.
- **Services Communication:** Services share schemas and contracts when they communicate, not internal classes.

Benefits Of Service-Oriented Architecture Style

The most important benefits of service-oriented architecture style are [Con+09]:

- **Domain Alignment:** Reuse of common services with standard interfaces increases business and technology opportunities and reduces cost.
- **Abstraction:** Services are autonomous and accessed through a formal contract, which provides loose coupling and abstraction.
- **Interoperability:** Because the protocols and data formats are based on industry standard, the provider and consumer of the service can be built and deployed on different platforms.
- **Discoverability:** Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.

Limits Of Service-Oriented Architecture Style

Service-oriented architecture is not suitable for [BA+15]:

- Applications which are based on heavy data exchange.
- Application that has short living time span.
- Application which does not provide full functionality or does not work as a complete system instead serve as a component and have limited scope.
- Application which is tightly coupled, where loose coupling is not recommended or to consider it would be pointless.

1.2.3 Software Architecture Importance

With a high level of abstraction, software architecture represents the structure of systems including its components, properties, and interaction with one another. Virtually, software architecture plays the role of a bridge between requirements and the implementation (code), so it can help in the following aspects of software development [Gar00]:

- **Construction:** Software architecture indicates the major system components, to allow developers to focus on its implementation, relationships, and iteratively to refine it.
- **Analysis:** Software architecture assists in the systems analysis aspect by system consistency checking, conformance to constraints imposed by an architectural style, conformance to quality attributes, dependence analysis, and domain-specific analyses for architectures built-in specific styles.
- **Understanding:** Presenting the software architecture for a large system with a high level of abstraction makes comprehend and understanding the system simple and easier.
- **Reuse:** The software architecture allows determining the reusable components of the system and the way to use them.
- **Management:** An evaluation of architecture leads to a much clearer understanding of requirements, implementation strategies, and potential risks to ensure successful development.
- **Evolution:** Software architecture provides the structure of the system and exposes the parts that need special attention for its evolution, this allows manage the propagation of changes and to evaluate the costs associated with the evolution.

1.3 Software Evolution

Evolution is an essential nature of software systems, and it is considered a process of progressive change in the properties of the evolving entity or that of one or more of its constituent elements[MFP06]. We present in this section the concept, the laws, and the process of software evolution and its importance.

1.3.1 Software Evolution Concept

Evolution of software recognized as one of the most problematic and challenging areas in the field of software engineering. Lehman et al.[LR02] describes software evolution phenomenon as following: *"Software evolution is the collection of all programming activities intended to generate a new version from an older and operational version."* It relates to how software systems change over time, and these changes may be in several types [Gri]:

- Changing the system so that it runs in a different environment from its initial implementation.
- Modifying the system to meet new requirements for adding new functionality.
- Changing the system to correct deficiencies to meet its requirements.
- Improving the performance and the structure of the system without changing functional behavior.

1.3.2 Software Evolution Laws

Lehman has given also laws for software evolution generally applicable to large, tailored systems developed by large organizations, they are [Gri; Tutb]:

- **Continuing Change:** Any software system that represents some real-world reality undergoes continuous change or become progressively less useful in that environment.
- **Increasing Complexity:** As the software system evolves, its structure becomes more complex unless effective efforts are made to avoid this phenomenon.
- **Conservation of Familiarity:** During the active lifetime of the program, changes made in the successive release are almost constant.
- **Continuing Growth:** The functionality provided by the systems must be continuously increased to maintain user satisfaction.
- **Declining Quality:** The quality of systems will decline unless modified to reflect changes in their operating environment.
- **Feedback Systems:** Evolution processes incorporate multi-agent, multi-loop feedback systems that must be treated as systems to achieve significant product improvement.
- **Large Program Evolution:** Program evolution is a self-regulating process. System attributes (such as size, the time between releases, and the number of reported errors...) are approximately invariant for each system release.
- **Organizational Stability:** Over the life of a program, its development rate is constant and nearly independent of the resource dedicated to developing the system.

1.3.3 Software Evolution Process

Software evolution processes differ depending on the type of software being maintained, the development processes used in an organization, and the skills of the people involved [Gri].

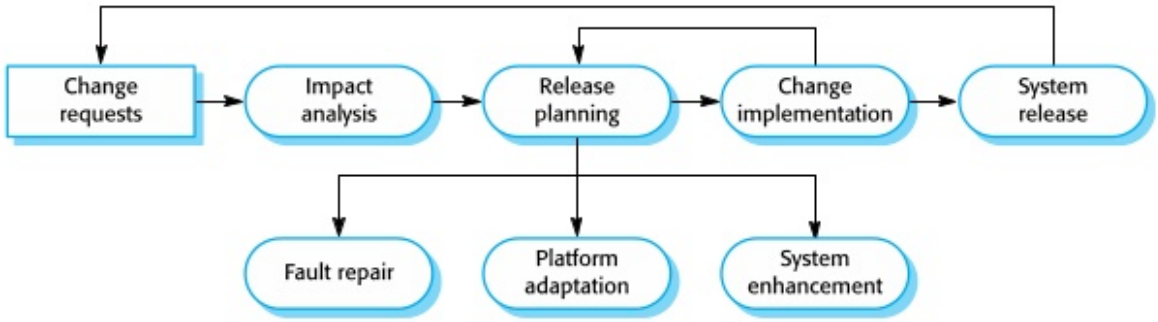

Figure 1.5: Software Evolution Process [Gri].

Figure 1.5 shows an overview of the software evolution process. It includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are evaluated to find out how much the system will be affected by the change and how much it will cost to implement the change. Planning for a new version of the system is linked to the acceptance of the proposed changes, with consideration to fault repair, adaptation, and new functionality. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release [Rah].

1.3.4 Software Evolution Importance

Software evolution is important because organizations have invested large amounts of money in their software and are now completely dependent on these systems. Their systems are critical business assets and they have to invest in system change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development [Gri].

1.4 Software Architecture Evolution

Software architecture evolution is the process of maintaining and adapting an existing software architecture to meet changes in requirements and environment instances of

industrial-scale architecture evolution often take months or years to complete and entail the expenditure of substantial resources.
In this section, we present the concept of software architecture evolution, its process, and its importance.

1.4.1 Software Architecture Evolution Concept

Jeffrey M.Barnes defined software architecture evolution as [Bar12]: *"Software architecture evolution is a phenomenon that occurs in virtually all software systems of significant size and longevity. As a software system ages, it often needs to be structurally redesigned to support new features, incorporate new technologies, adapt to changing market conditions, or meet new architectural quality requirements. In addition, many systems over the years tend to accrue a patchwork of architectural workarounds, makeshift adapters, and other degradations in architectural quality, requiring some sort of architectural overhaul to address."* Architecture evolution can occur at the specification time that called a static evolution or at the system execution time that called dynamic evolution.

- **Static Evolution:** Static evolution is principally concerned with structuring systems as separated abstractions, and then evolving these abstractions offline. By identifying all dependencies, and isolating those abstractions that are independent, static evolvability enables individual application requirements to be changed in isolation [Fal+04].

- **Dynamic Evolution:** Dynamic evolution means that software architectures can be modified and those changes can be enacted during the system's execution. This behavior is most commonly known as runtime evolution or reconfiguration. The typical evolution operations for the dynamic evolution of software architectures include adding, removing, and updating components or connectors, changing the architecture topology by adding or removing connections between components and connectors [XZ10].

Evolution of software architecture can be affect by some architectural dimensions [Lea]:

- **Technical:** The implementation parts of the architecture (the frameworks, dependent libraries) and the implementation language(s).

- **Data:** Database schemas, table layouts, optimization planning.

- **Operational/System:** Concerns how the architecture map to existing physical and/or virtual infrastructure (servers, switches, cloud resources).

1.4.2 Software Architecture Evolution Process

The software architecture evolution process is a plan as a series of evolution states and evolution transitions leading from the initial architecture to the target architecture. An evolution path is a plan described in such a way. Figure 1.6 shows an evolution process graph of software architecture. In which, the software architecture is the node in the graph and the possible evolutionary transitions are the edges.

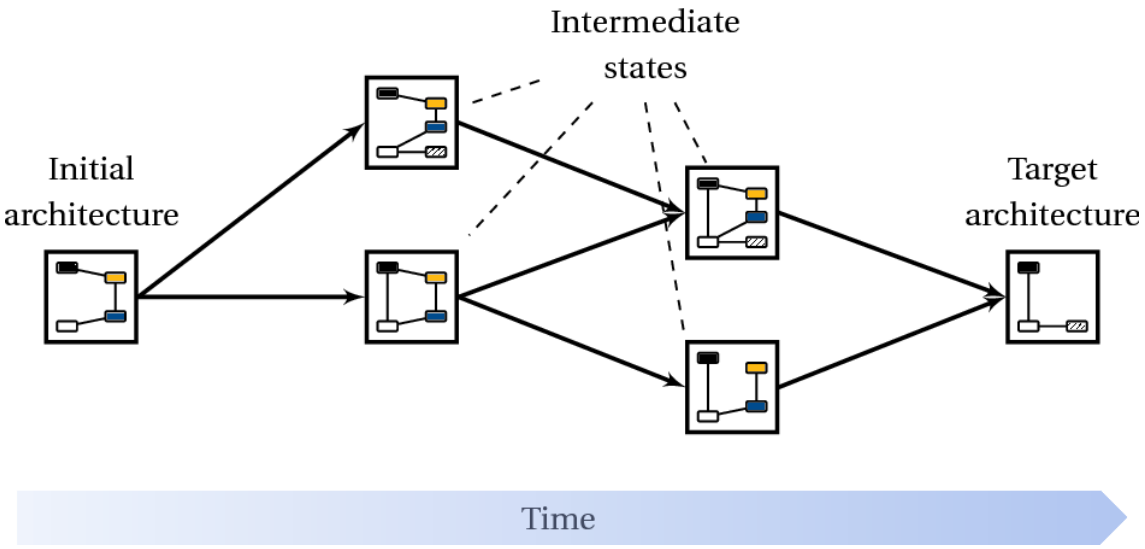

Figure 1.6: Software Architecture Evolution Process Graph [Bar13].

In any evolution possible scenario, there may be many evolution paths that seem feasible. The task of the evolution architect is to consider these paths and select the one that is optimal of evolving the system from the initial architecture to the target architecture.

1.5 Conclusion

In this chapter, we talked about the terminology of software architecture, and the most famous styles of software architecture, client-server architecture (CSA), object-oriented architecture (OOA), component-based architecture (CBA) and service oriented architecture (SOA) with main principles, benefits and limits of every style. In this work, we adopt the component-based architecture style. Also, we talked about software architecture importance. Then, we presented the concept, the eight Lehman's laws of software evolution, law of continuing change, increasing complexity, conservation of familiarity, continuing growth, declining quality, feedback systems, large program evolution, and law of organizational stability. Also, we took an overview of software evolution process

and its importance. In the last part, we talked about software architecture evolution concept and their kinds, static and dynamic evolution. Also, we talked about how the evolution of software architecture can be affect by technical or data side. Then we took an overview of its process. In this work, we are more interested in the dynamic evolution of software architecture.

Chapter 2

Component-Based Software Development

2.1 Introduction

The primary objective of component-based software development is to ensure component reusability. A component encapsulates functionality and behaviors of a software element into a reusable and deployable binary unit.
In this chapter, we present the software component and component model concepts and take an overview of the different components technologies or models. Then, we introduce the Open Service Gateway Initiative technology (OSGi), one of the component models. Also, we present its components, services, and what are its benefits.

2.2 Software Component Definition

There is tow definitions of a software component, the first one focuses on characterization of a software component of Szyperski [Szy02]: *"A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party."* The second definition of Heineman and Councill is more general [HC01] : *"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."* From this tow definitions, we conclude:

- A component is a composition unit.
- A component can be deployed on different platforms.

- A component may be a subject to composition by a third party.
- A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities.
- A component has an obviously defined interface and conforms to a recommended behavior common to all components with an architecture.

2.3 Component Model Definition

Heineman and Councill define a component model as follows [HC01]: *"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment."* From this definition, a component model is the set of rules and abstraction that allow the characterization, implementation and deployment of software components. There are tow component model types, flat component models and hierarchical component models.
The hierarchical model is more flexible and extensible than the flat model, which is indeed a particular case of hierarchical model. This type of model have another property, it is possible to compose new components from existing native components, also it support declarative specification of composite components [AME12].

2.4 Component Models

There are a lot of component model, in this work we just present five models, CORBA Component Model (CCM), Enterprise Java Beans (EJB), Fractal Component Model, the Open Services Gateway Initiative (OSGi) and Service Component Architecture (SCA).

2.4.1 CORBA Component Model

Common Object Request Broker Architecture or CORBA component model (CCM) is a basic model of OMG's (Object Management Group) component specification. The CCM specification defines an abstract model, a programming, a packaging model, a deployment model, an execution model and a meta-model defines the concepts and the relationships of the other models [Crn+11].

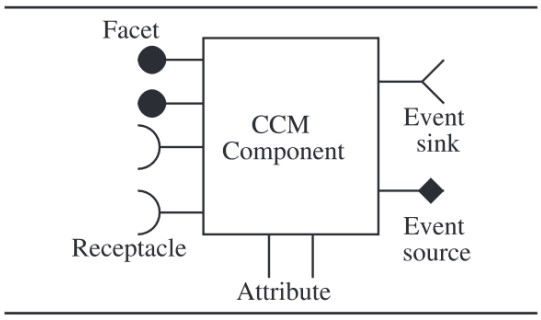

Figure 2.1: A CCM Component [PPR03]

A component in CCM is composed of a set of ports, through this ports component communicate with outside world.
Figure 2.1 represent a component of CCM and the kinds of ports. There are five kinds of ports [PPR03]:

1. **Facets:** are named connection point that provide services available as interfaces.
2. **Receptacles:** are named connection point to be connected to a facets. They describe the component's ability to use a reference supplied by some external agent.
3. **Event Source:** are named connection points that emit typed events to one or more interested event customers, or to an event channel.
4. **Event Sinks:** are named connection points into which events of a specified type may be pushed.
5. **Attributes:** are named values exposed through read and write operations.

CCM uses a separate language IDL (Interface Definition Language) for the component specification, also provides a Component Implementation Framework (CIF) which relies on Component Implementation Definition Language (CIDL) and describes how functional and nonfunctional parts of a component should interact with each other [Crn+11].

2.4.2 Enterprise Java Beans

Enterprise Java Beans (EJB) developed by Sun MicroSystems envisions the construction of object-oriented and distributed business applications. It provides a set of services, such as transactions, persistence, concurrency and interoperability.

EJB is an architecture for setting up program components, written in the java programming language, that run in the server parts of a computer network that uses the client/server model [Rou], also it is like CORBA component model uses a flat component model [Crn+11].

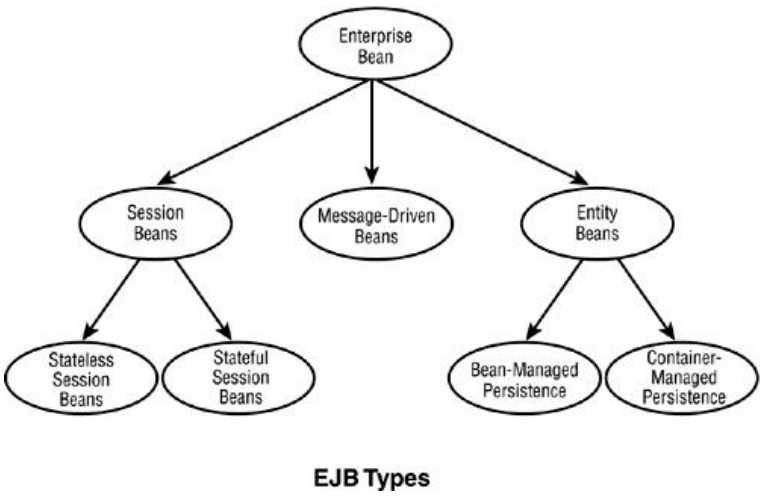

Figure 2.2: Enterprise Java Beans Types

EJB offer three types of components as 2.2 shown, and they namely:

- **Session Bean:** It implement a conversation between a client and the server side. Also, it execute a particular business task on behalf of a single client during a single session. There are tow types of session bean [WOR; Tuta]:

  1. Stateful Session Bean: A stateful bean represents a conversational session with a particular client.
  2. Stateless Session Bean: A stateless bean represents a conversation with the client without storing any state.

- **Entity Beans:** Entity beans are persistent objects. They typically represent business entities, such as customers, products, accounts and orders. Each entity bean has an underlying table in a relational database. The state of an entity bean is persistent, transactional and shared among different clients. There are tow types of entity beans:

  1. Container-Managed Persistence (CMP): In a CMP entity bean, the EJB container manages the bean's persistence according to the data-object mapping in the deployment descriptor. Any change in the entity bean's state will be automatically saved to the database by the container.

2. Bean-Managed Persistence (BMP): A BMP entity bean has to manage both the database connections and all the changes to the bean's state.

- **Message Driven Beans:** Message driven beans are used in context of JMS (Java Messaging Service). It can consumes JMS messages from external entities and act accordingly.

Each of these beans is deployed in an EJB container which is in charge of their management at runtime (start, stop, passivation or activation).

2.4.3 FRACTAL

FRACTAL is an hierarchical component model developed by France Telecom, R&D and INRIA [Crn+11]. It is a general component model which is intended to implement, deploy and manage complex software systems, including in particular operating systems and middleware. The main characteristic of fractal model are [Bru+04]:

- **Introspection Capabilities:** Introspection capabilities to monitor a running system.
- **Re-configuration Capabilities:** To deploy and dynamically configure a system.
- **Composite Components:** To have a uniform view of applications at various levels of abstraction.
- **Shared Components:** To model resources and resources sharing while maintaining component encapsulation.

FRACTAL is defined as an extensible system of relations between selected concepts, where components can be endowed with different forms of reflective features **"control"**.The following are examples of useful forms of controllers [Bru+04]:

- **Attributes Controller:** An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.
- **Binding Controller:** A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces.
- **Content Controller:** A component can provide the ContentController interface to list, add and remove sub-components in its contents.

- **Life-cycle Controller:** A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration, this interface include methods to start and stop the execution of the component.

An interface is an access point to a component, that supports a finite set of operations. There are tow kinds of interfaces [Bru+04]:

- **Service Interfaces:** Which this interfaces correspond to access points accepting incoming operation invocations.
- **Client Interfaces:** Which this interfaces correspond to access points supporting outgoing operation invocations.

The main purpose of Fractal is to provide an extensible, open and general component model that can be tuned to fit a large variety of applications and domains.

2.4.4 Open Services Gateway Initiative

Open Services Gateway initiative (OSGi) is a set of specifications that define a dynamic component system for Java [Allc], in which these specifications enable components to hide their implementations from other components while communicating through services, which are objects that are explicitly shared between components [Alla]. OSGi component is named bundle or plugin, interact with each other via an Application Programming Interface (API). The API is defined as a set of classes and methods which can be used from other components. A component also has a set of classes and methods which are considered as internal to the software component.
The components can be dynamically installed, activated, de-activated, updated and uninstalled [Gmb]. The OSGi specification has several implementations, for example Eclipse Equinox and Apache Felix. In the next section, we will talk about OSGi with more detail.

2.4.5 Service Component Architecture

Service component architecture (SCA) is a set of specifications released by the OSOA (Open Service Oriented Architecture), which describe a model for building applications and systems using a Service Oriented Architecture (SOA). SCA is based on the idea that business function is provided as a series of services, which are assembled together to create solutions that serve a particular business need. Also, SCA provides a model

both for the composition of services and for the creation of service component including the reuse of existing application function within SCA compositions [Geya].

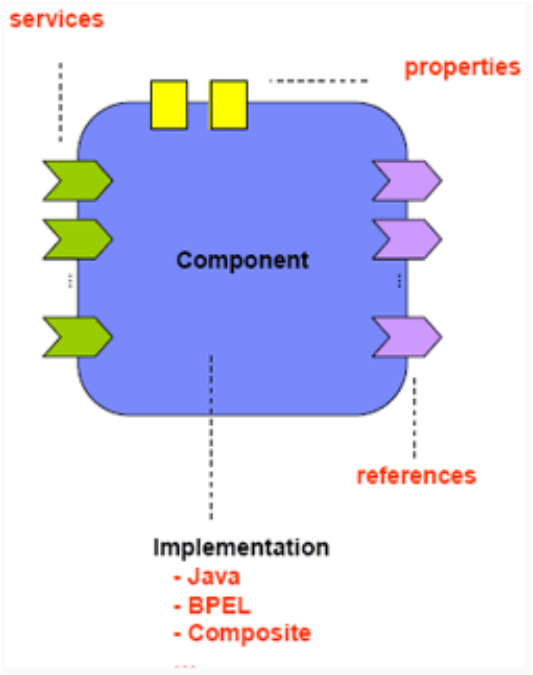

Figure 2.3: SCA Component [Haa].

Figure2.3 shown the structure of a SCA component, and it consists of [Geyb]:

- **Service:** It represent an addressable interface of the component.
- **Reference:** It represent a requirement that the component has a service.
- **Properties:** A property allows for the configuration of component with externally set values.
- **Implementation:** It represent characteristics inherent to the component itself.

2.5 Open Services Gateway Initiative Technology

Open Service Gateway Initiative or OSGi is a specification defining a Java based component system. The first version of OSGi dates in Mars 1999 by the OSGi Alliance (which includes IBM, Oracle, Samsung, Nokia, $\cdots$) [Tib]. The OSGi Alliance's framework specification defines the proper behavior of the framework, which gives a well-defined API to program against [Hal+11].

2.5.1 OSGi Framework

The OSGi framework has a central role in OSGi-based applications creation considering it's the application's execution environment [Hal+11], that is means the framework is the runtime that implements and provides OSGi functionality. The functionality of the Framework is divided in the following layers as the figure 2.4 is shown:

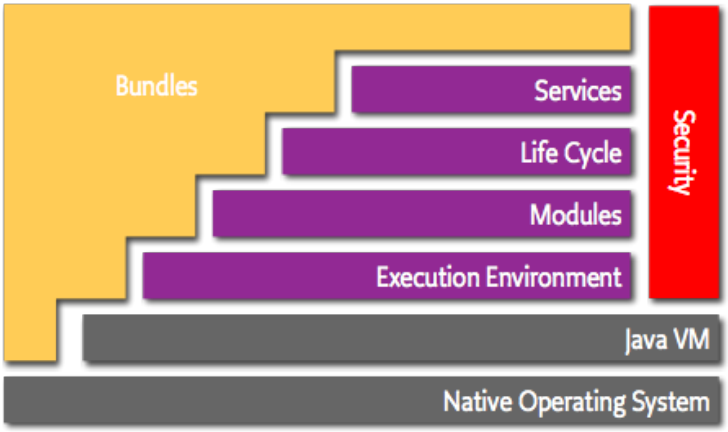

Figure 2.4: Layering-OSGi [Allc].

- **Bundles:** Bundles are the name of OSGi components, also called plugins.
- **Security:** The security layer is an optional layer that underlies the OSGi framework that handles the security aspects.
- **Services:** The service layer concerned with interaction and communication among modules, specifically the components contained in them.
- **Life-Cycle:** This layer concerned with providing execution-time module management and access to the underlying OSGi framework.
- **Modules:** This layer that defines how a bundle packaging and sharing code.
- **Execution Environment:** The execution environment layer defines the available methods and classes in a specific platform.

2.5.2 OSGi Framework Implementations

The OSGi specification enables the creation of multiple implementations of the core framework and that by several open-source and commercial entities [MAV10]. There are four open-source implementations of the framework are the most famous [Allc]:

- **Apache Felix:** Felix developed by Apache[1] and it provides framework implementation in addition to many service implementation.
- **Eclipse Concierge:** Concierge[2] is an optimized OSGi framework implementations, it ideal for mobile or embedded devices. It developed as part of the flowSGi project, which is an ongoing research project at Institute for Pervasive Computing, ETH Zurich.
- **Eclipse Equinox:** Equinox is the reference implementation for the core framework specification and several service specifications, developed by IBM and supplied under the Eclipse Public License (EPL[3]). It is the base runtime for all Eclipse tooling, rich client and service-side.
- **Knopflerfish:** Knopflerfish[4] provides framework implementation, open-source OSGi SDK and runtime container. Led and maintained by Makewave.

2.5.3 OSGi Component

The OSGi specification defines a component model to be executed in the framework, called bundle or plugin in Eclipse terminology.

2.5.3.1 Bundle Concept

Bundle is a JAR file with extra meta-data (META-INF/MANIFEST.MF), class files and their related resources (plugin.xml) as the figure 2.5 is shown.
There are two things that make bundles stronger than standard JAR files [Hal+11]:

- Ability to declare on which external packages the bundles depend and they called imported packages.
- Ability to declare which contained packages are externally visible and they called exported packages.

[1] Apache: `http://felix.apache.org/`
[2] Concierge: `https://www.eclipse.org/concierge/`
[3] EPL: `https://www.eclipse.org/equinox/`
[4] Knopflerfish: `https://www.knopflerfish.org/`

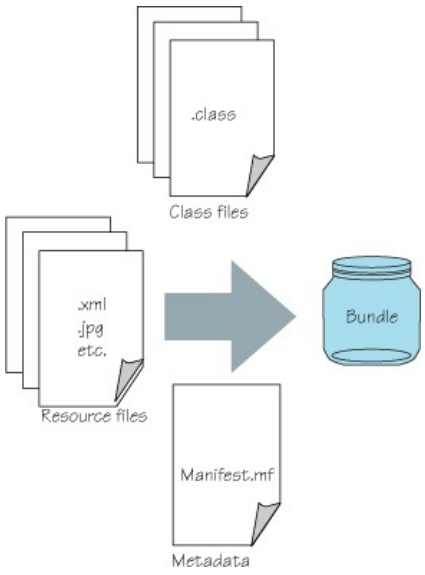

Figure 2.5: Bundle Contents [Hal+11].

Meta-Data File

The meta-data file specified enables the OSGi Framework to process the modular aspects of the bundle, and it called META-INF/MANIFEST.MF file [Cen]. Figure 2.6 shows an example bundle manifest file (META-INF/MANIFEST.MF).

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: %pluginName
Bundle-SymbolicName: org.archbasedevol.dynarch;singleton:=true
Bundle-Version: 0.1.0.qualifier
Bundle-ClassPath: .
Bundle-Vendor: %providerName
Bundle-Localization: plugin
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: newcompo,
 newcompo.impl,
 newcompo.util,
 org.archbasedevol.dynarch.activateur
Require-Bundle: org.eclipse.core.runtime,
 org.eclipse.emf.ecore;visibility:=reexport,
 org.eclipse.osgi
Bundle-ActivationPolicy: lazy
Bundle-Activator: org.archbasedevol.dynarch.activateur.Activateur
Import-Package: org.eclipse.emf.ecore.xmi,
 org.eclipse.emf.ecore.xmi.impl
```

Figure 2.6: Example Bundle MANIFEST.MF file.

The most important information can be found in the bundle manifest file:

- **Bundle-ManifestVersion:** Indicated the OSGi specification to use for reading this bundle.
- **Bundle-Name:** A name was given by the developer.

- **Bundle-SymbolicName:** Bundle name (unique).
- **Bundle-Version:** The version of bundle.
- **Bundle-ClassPath:** Specifies where to load the bundle classes, default is (.).
- **Bundle-RequiredExecutionEnvironment:** Specify the version of Java required to run the bundle.
- **Export-Package:** The visible packages are declared out of the bundle.
- **Require-Bundle:** The list of required bundles.
- **Bundle-ActivationPolicy:** Allow a bundle to specify a boot strategy.
- **Bundle-Activator:** Define the name of the executable class in the bundle.
- **Import-Package:** The external dependencies of the bundle that the OSGi Framework uses to resolve the bundle.

Resources File

From the resource file, we take plugin.xml, describes how the bundle extends the platform, what extensions it publishes itself, and how it implements its functionality [FOU]. In a separate Java JAR file can found the implementation code, is loaded when the bundle has to be run. Figure 2.7 example bundle plugin.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>

<!--
-->

<plugin>

   <extension point="org.eclipse.emf.ecore.generated_package">
      <!-- @generated newcompo -->
      <package
            uri="platform:/resource/org.archbasedevol.dynarch/model/newcompo.ecore"
            class="newcompo.NewcompoPackage"
            genModel="model/newcompo.genmodel"/>
   </extension>

</plugin>
```

Figure 2.7: Example Bundle "plugin.xml" file.

2.5.3.2 Bundle Lifecycle

The lifecycle explains how bundles management dynamically in the OSGi framework. Figure 2.8 shows the possible bundle states transitions.

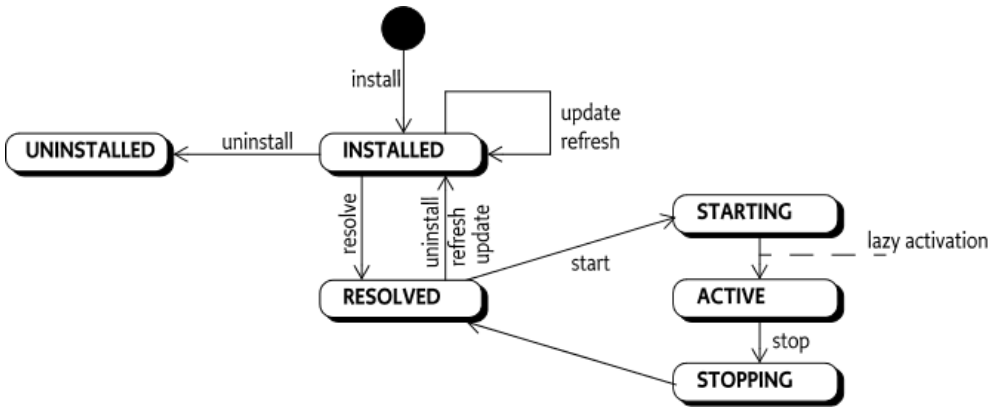

Figure 2.8: Bundle Lifecycle [Allb]

The bundle can be in one of these six states [Allb]:

- **ACTIVE:** A bundle is in ACTIVE state meaning is running now.
- **RESOLVED:** A bundle is in RESOLVED state meaning is ready to be started or has stopped.
- **INSTALLED:** A bundle is in INSTALLED state meaning is installed in the framework but is not yet resolved.
- **STARTING:** A bundle is in STARTING state meaning is in the process of starting, the bundle will stay in the STARTING state until the bundle is activated.
- **STOPPING:** A bundle is in STOPPING state meaning is in the process of stopping and moving to the RESOLVED state.
- **UNINSTALLED:** A bundle is in UNINSTALLED state meaning can not move into another state and may not be used.

The bundle changes its state automatically these can be safely added to and removed from the framework without restarting the application process or by the developer manually.

2.5.4 OSGi Services

OSGi services are Java interfaces representing a conceptual contract between service providers and service clients [Hal+11]. The services are registered in the Service Registry so that they can use or consume through other bundles.

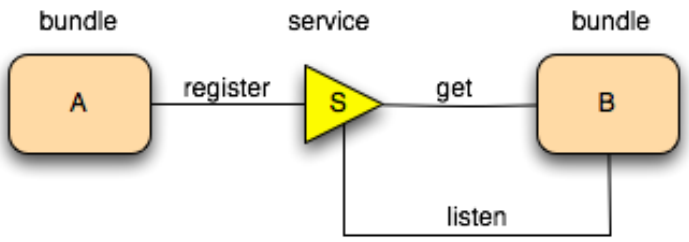

Figure 2.9: Operation of OSGi Services [Allc]

Figure 2.9 show how OSGi services work, so that a bundle "A" can register/publish a service implementation or multiple services at the same time, another bundle "B" can consume it, and that by finding and getting this service(s) from the OSGi Service Registry.

2.5.4.1 Service Component

A Java class inside a bundle that is declared via component description in an XML document and managed by a Service Component Runtime [Blo]. The Service Component Runtime or SCR is an implementation of the OSGi declarative services specification offering a service-oriented component model to simplify OSGi-based development [Ale]. There are three types of component [Blo]:

- **Delayed Component:** A Delayed Component needs to specify a service and activation does not occur until a service object is requested. Therefore, class loading and copying can be delayed until that time.
- **Immediate Component:** An Immediate Component does not need to specify a service and they are activated as soon as their dependencies are satisfied.
- **Factory Component:** A Factory Component creates and activates new Component Configurations on request.

2.5.4.2 Service Component Lifecycle

Service Components have their own lifecycle, which is contained in the bundle lifecycle (see Figure 2.8).
Figures 2.10 and 2.11 shows the possible Delayed/Immediate Component states transitions.

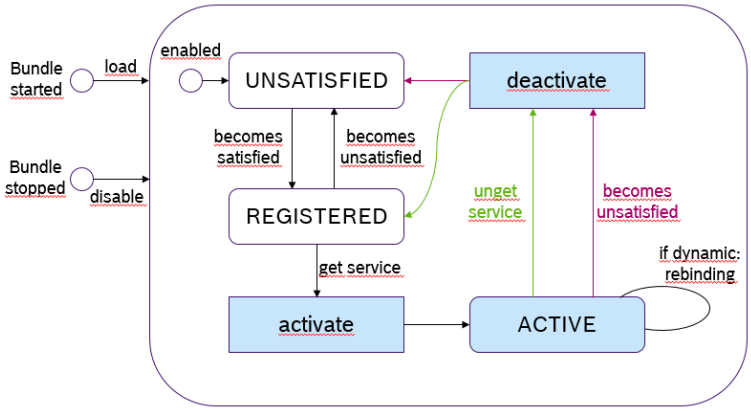

Figure 2.10: Delayed Component Lifecycle [Blo]

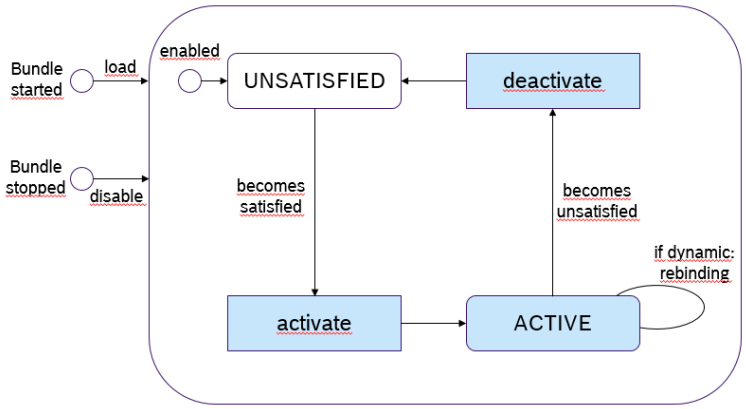

Figure 2.11: Immediate Component Lifecycle [Blo]

From these two figures, the possible components states are:

- **Disabled:** All components are disabled when the bundle is stopped.
- **Enabled:** The component's initial enabled state is defined by the component description.
- **Satisfied:** The component is in a satisfied state when it is enabled and the required reference services are available.
- **ACTIVE:** The component has been activated due to immediate activation or, in case of a Delayed Component, it was requested.
- **REGISTERED:** Only the Delayed Components can be in the REGISTERED state if it satisfied and not yet requested.

- **UNSATISFIED:** The component is not ready to start or is not satisfied anymore.

2.5.5 OSGi Benefits

OSGi provides a stable and evolving technology platform for developing open-source projects with a set of benefits [Alld]:

- **Reduced Complexity:** OSGi bundles hide their internals from other bundles and just communicate through defined services, that reduce the number of bugs and make bundles simpler to develop through defined interfaces.
- **Dynamic Updates:** Dynamic OSGi component model allows bundles to change their states (installed, started, stopped, updated, and not installed) without bringing down the whole system.
- **Reuse:** The OSGi component model allow easily to use other components in an application.
- **Real World:** Dynamic OSGi framework allows the update of bundles quickly with a dynamic services movement that makes the services a perfect match for many real world scenarios.
- **Simple:** Despite the powerful dependency management, configuration, and dynamics, OSGi code looks almost identical to classic Java code.
- **Runs Everywhere:** This can only be true if:
  1. OSGi APIs should not use classes not available in all environments.
  2. The bundle should not start if it contains a code that is not available in the execution environment.
- **Non Intrusive:** This strategy makes application/bundle code easier to port to another environment.
- **Secure:** The OSGi framework is loading the classes from bundles and it knows for each bundle exactly which bundle provides these classes, which is a significant speed up factor at startup.

2.6 Conclusion

In this chapter, we presented the different concepts of the software component and the component model. Also, we took an overview of the most famous component models, started with Common Object Request Broker Architecture Component Model (CORBA), Enterprise Java Beans (EJB), FRACTAL, Open Services Gateway Initiative (OSGi) and Service Component Architecture (SCA).
In this work our interested in OSGi technology, so we presented more detail and explanation on this technique. First, the OSGi framework and the famous implementations of this framework, like Apache Felix, Eclipse Concierge, Eclipse Equinox, and Knopflerfish. Then we talked about bundle (or plugin) the OSGi component and what it's contents(meta-data file, resources file and class files) and it's lifecycle. Also, we took an overview of OSGi Services, it's components and lifecycle. Finally, we presented OSGi benefits that make it a successful technology in the dynamic module system.

Chapter 3

State of the Art

3.1 Introduction

We present in this chapter the state of the art of dynamic software architecture evolution approaches. In this work, we are interested in dynamic software evolution for component/service-oriented systems. First, we present works about software architecture recovering (component-based and service-oriented architecture). Then, we talk about existing works of software architecture static and dynamic evolution.

3.2 Component-Based Architecture Recovering

There is many existing component-based architecture recovering works in the literature such as [LZN04; Cha+08; ESH10; ML16; Lut+17] in which show a different proposed approaches.

Chung-Horng Lung et al., [LZN04] presented an approach that can be applied to software partitioning, recovery, restructuring, and decoupling. This approach can support a rapid and effective evaluation of a system based on the relationships between components and features, or component interdependencies at various levels of abstraction. System partitioning is usually performed by designers based on their experiences. Also, it can help designers quickly obtain an outline of the architecture or design and provide an alternative view. More evaluations could then be conducted to identify potential problems early in the development process. The architecture recovered from the approach, if different from the current or existing design, could also force the designers to reason and compare the differences, and potentially restructure the system.

Sylvain Chardigny et al., [Cha+08] proposed an approach called ROMANTIC for the extraction of a component-based architecture from an object-oriented system. The main idea of this approach is to propose a quasi-automatic process of architecture recovery based on semantic and structural characteristics of software architecture concepts. These characteristics guide the partitioning of the system classes and the abstraction of each shape in a component. The extraction process split into two steps, the first one is to define a correspondence model between code elements (that is mean object concepts such as classes, interfaces, packages, ...) and the architectural concepts (components, connectors, interfaces, ...). The second is to instantiate the previous correspondence model and extract the architecture from the software, in which that architecture must be respects the four guides: i) must be semantically correct. ii) must have good quality properties. iii) must respect precisely the recommendation of the architect, and specifications and constraints defined in documentations. iv) must be able to be adapted to the specificity of the deployment hardware architecture).

Alae-Eddine El Hamdouni et al., [ESH10] proposed an approach of architecture recovery which aims to extract component-based architecture from an object-oriented system, by a semiautomatic exploration process in order to identify the architectural components by the relational concept analysis (RCA). The RCA approach comes as a complementary method to relieve some limits of the existing implementation of ROMANTIC based on a simulated annealing algorithm. The architectural components in this approach are identified from concepts derived by exploiting all existing dependency relations between classes of the object-oriented system. The relational concept analysis process steps are: i) extraction of a dependency graph (DG) of the source code classes. ii) creation of an RCA model using the information of the dependency graph. iii) generation of a lattice of concepts representing clusters of object classes. iv) identification of candidate components from resulting lattice. The obtained lattice allows to: i) identify architectures with several abstraction levels. ii) identify composite components. iii) select components according to some grouping criteria and navigating in the lattice. The evaluation of the feasibility of the approach was on a Java software.

Hong Mei and Jian Lü [ML16] presented an approach to recovering software architecture from component-based systems at runtime and changing the runtime systems via manipulating the recovered software architecture in which can accurately and thoroughly describe the actual states and behaviors of the runtime system. In order to keep the recovered software architecture update at any time and change the runtime system via manipulating its recovered software architecture, the elements in the recovered software architecture are implemented as a set of meta-objects that are created at runtime

and reflect other runtime entities internal of the system. It ensures that changes made on the recovered software architecture immediately lead to corresponding changes in the actual states and behaviors of the runtime system, and vice versa. Manipulation of the recovered software architecture can be divided into the next categories: i) lifecycle management of runtime entities. ii) add/remove/replacement of runtime entities. iii) statistics of runtime entities. iv) business invocation of methods exposed by runtime entities. The approach presented is demonstrated on Peking University Application Server (PKUAS), a reflective J2EE (Java 2 Enterprise Edition) application server.

Thibaud Lutellier et al., [Lut+17] worked to improve on previous studies of y recover software architectures from software implementations by study the impact of leveraging accurate symbol dependencies on the accuracy of architecture recovery techniques. In addition, they evaluated other factors of the input dependencies such as the level of granularity and the dynamic-bindings graph construction, in which they evaluated nine architecture recovery techniques. The results suggest that: i) using accurate symbol dependencies has a major influence on recovery quality. ii) more accurate recovery techniques are needed. Also, they developed a new submodule-based technique to recover preliminary versions of ground-truth architectures.

3.3 Service-Oriented Architecture Recovering

In this section, we present an existing service-oriented architecture recovering works in the literature such as [GSK14; ASS14; KTS18].

Marvin Grieger et al., [GSK14] proposed a semi-automatic approach that combines hierarchical and partitioning clustering in order to improve initial service design by restructuring the service-oriented architecture. The purpose of the approach is to create a maintainable, service-oriented architecture. There are two steps in the restructuring process. The first step is to perform hierarchical clustering based on the dependencies between the services. The resulting clusters are aggregate services that are related in terms of the underlying business process. The second step of the restructuring process is to detect and remove software clones. The described process has been manually applied to an application for architecture reconstruction and migration of an enterprise legacy system.

Seza Adjoyan et al., [ASS14] proposed an approach that identifies services as groups of classes in the legacy software source code, it based on the definition of a fitness function that measures semantic the correctness of each group of source code elements

to be considered as a service. In order to identify the services from object-oriented code, a mapping between object-oriented and service-oriented architecture concepts is defined. The proposed approach has been evaluated on just two realistic case studies Java Calculator Suite and Mobile-Media.

Mohamed Lamine Kerdoudi et al., [KTS18] proposed an approach to recover the architecture of a service-oriented system, depending on a particular use case that reflects the use context. This approach contributes to a multi-step process that analyzes the source code of the system and interacts with the runtime environment, including the service registry, to build a first core architecture modeling the components of the system that always run then this core architecture is enriched. In this approach process, a method to reduce the size and complexity of the architecture in which when the architectures recovered from large systems are however complex and difficult to grasp or to grip by spotlight the active elements (components/services) and ignore other elements that are not running in which represent noise in the recovered architecture. The implementation of the work proposed process on the OSGi system, in which the OSGi platform provides a well-know service-based framework for Java applications.

3.4 Software Architecture Static Evolution

In this section, we present an existing software architecture static evolution works in the literature such as [OST05; GS09; BGS14]

Mourad Oussalah et al., [OST05] proposed a model for software architecture evolution, independently of their description language, named SAEV (Software Architecture EVolution Model. The SAEV offered a set of concepts to describe and manage the evolution of a given architecture, that evolution is reflected through the different changes carried out on the software architecture elements such as adding or removing components (elements). The SAEV associates an evolution strategy for each architectural element. A strategy gathers the whole of evolution rules which describes the operations that can be applied to this architectural element. SAEV proposed also an evolution mechanism in which describes the execution process of the evolution model at specification time and at execution time.

David Garlan and Bradley Schmerl [GS09] developed a tool for architecture evolution planning and analysis that allows architects to plan evolutionary changes to a software system from an architectural perspective. Architects can define changes to be made in each step of evolution and can explore multiple such evolution paths. The tool

provides a plugin framework allowing analyses so that an architect can compare and swap multiple possible evolution paths. These analyses can be tailored to particular evolution domains and to particular business environments of concern to the architect.

Jeffrey Barnes et al., [BGS14] described an approach for planning and reasoning about architecture evolution. The approach focuses on providing architects with the means to model prospective evolution paths and supporting analysis to select among these candidate paths. They characterized recurring patterns as a set of related paths, which we term evolution styles. Such styles can be formally characterized, allowing for support by tools. They evaluated their approach for software architecture evolution modeling by two very different methods. In the first method, they evaluated the computational complexity of model-checking evolution path constraints. This theoretical study showed that the approach to verifying path validity is computationally feasible. In the second method, they demonstrated the applicability of the approach in a small case study that exemplifies the kinds of concerns that arise in real world evolution.

3.5 Software Architecture Dynamic Evolution

In this section, we present an existing software architecture dynamic evolution works in the literature such as [HMY04; FA04; Pér+05; OST05; ML16; HQO16]

Gang Huang et al., [HMY04] proposed runtime software architecture (RSA) supporting architecture-based software maintenance and evolution. RSA extends the traditional architecture description language to accurately represent the actual states and behaviors of the runtime system according to the perspective of software architecture at runtime. RSA can immediately capture changes of the runtime system so as to keep itself updated and ensure that changes made on itself will immediately lead to corresponding changes of the runtime system. The approach presented is demonstrated on Peking University Application Server (PKUAS), a reflective J2EE application server. A set of APIs is defined for accessing and manipulating RSA, are protected by some access control mechanisms. There is also a PKUAS Management Tool, a graphical maintenance and evolution tool built on RSA.

Paolo Falcarin and Gustavo Alonso [FA04] proposed an approach to dynamically evolve a software architecture based on runtime aspect-oriented programming. They developed also a framework called JADDA (Java Adaptive component for Dynamic Distributed Architectures) that allowed a system designer or administrator can control the architecture of an application by dynamically inserting and removing code extensions.

The JADDA aims to: i) easing timeline variability (changes that can be applied at either development time or run time) of different middleware implementations used. ii) updating the connectors of a system by acting on its xADL (XML-based Architecture Description Language) architectural specification, in this way reconfiguration can be decided at a higher level than source code and all the needed information are stored in the xADL file. iii) dynamic reconfiguration is also handled using a dynamic programming side platform to allow additional classes transport in a reliable. The work focused on handling more complex architectures with related consistency issues.

Jennifer Pérez et al., [Pér+05] presented a solution to the evolution problem of software architectures provided by PRISMA. PRISMA is an architecture modeling approach that integrates the advantages of component-based software development and aspect-oriented software development, it presented as a framework to evolve aspect-oriented and component-based architectures by requirements driven evolution. The evolution is supported by means of a meta-level and the reflexive properties of PRISMA which have been implemented as middleware. It is demonstrated also how the evolution services of the PRISMA meta-level permit the run-time evolution of software architectures using an industrial case study, the TeachMover Robot.

Mourad Oussalah et al., [OST05] also proposed a model SAEV for software architecture evolution, independently of their description language, that we talked about it int the previous section.

Hong Mei and Jian Lü [ML16] presented an approach to recovering software architecture from component-based systems at runtime and changing the runtime systems via manipulating the recovered software architecture in which can accurately and thoroughly describe the actual states and behaviors of the runtime system, we had mentioned the details of this work in the section.

Adel Hassan et al., [HQO16] proposed a dynamic evolution style for the software architecture dynamic evolution, and to provide a style sufficiently rich to model the dynamic changes in the software architecture of a real-time system and to be able to represent the potential ways of performing these changes. For that, they integrated the behavior concepts of dynamic changes into the Meta Evolution Style (MES) so they can have a sound understanding of dynamic evolution issues (safe stopping of running artifacts, transferring state, change management, and dynamic evolution scheduling) and constraints, which is a prerequisite to developing a modeling environment that supports dynamic evolution styles.

3.6 Conclusion

In this chapter, we presented the state of the art of dynamic software architecture evolution approaches. We first presented some works of software architecture recovering (component-based and service-oriented architecture). Then we presented works of software architecture static and dynamic evolution. In this work, we focus on dynamic software evolution for component/service-oriented systems.

Chapter 4

Software Architecture-Based Evolution for Component/Service Oriented Systems

4.1 Introduction

In this chapter, we present our proposed approach for software architecture based evolution of component/service-oriented systems. We start first with an overview of the proposed approach, then we give more details of each step. We present also in this chapter our proposed OSGi meta-model with an explanation of all its elements.

4.2 General Process

We first introduce an overview of the proposed approach for software architecture based evolution for component/service-oriented systems, then we present our proposed OSGi meta-model.

4.2.1 Proposed Approach Overview

Figure 4.1 depicts the overall process of our approach of the software architecture based evolution for component/service-oriented systems. It is divided into two main sub-processes, the first one titled *Software Architecture Recovering*. Its input is the application at runtime in different use cases, and gives as output component-based architecture after going through a few steps.
The second process named *Dynamic Software Architecture Evolution*. It takes as input

existing software architecture or component-based architecture and gives a new software architecture (updated version) and an evolved system after going through a few steps.

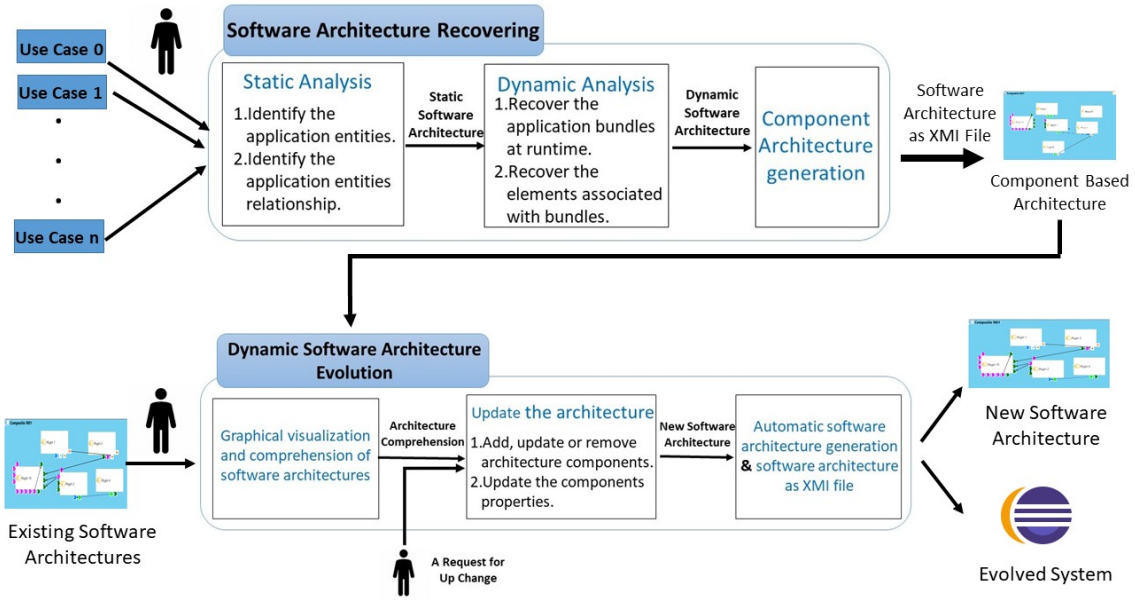

Figure 4.1: Our Proposed Approach for Software Architecture Based Evolution for Component/Service Oriented Systems.

For each process, there are certain steps:

1. **Software Architecture Recovering**:

   - Static Analysis: As input, this step takes the application at runtime in different use cases to analyze and identifies the system entities and their relationship with each other. The output is static software architecture.
   - Dynamic Analysis: In this step, the software architecture recovering at runtime task is starting by recovering the application bundles and the associated elements (components) with these bundles. The output is dynamic software architecture.
   - Component Architecture Generation: In the final step, creating the XMI file version of the recovered software architecture at runtime and generating the component model in which it is an instance from our meta-model. The output of this step is the result of the *Software Architecture Recovering* process, and it is a component based architecture.

2. **Dynamic Software Architecture Evolution**:

   - Graphical Visualization and Comprehension of Software Architectures: As an input, it can be an existing software architecture or component based architecture (the previous process result) to visualize the architecture graphically and to make its comprehension easier. The output is a graphical visualization of software architecture.
   - Update the Architecture: The graphical visualization allows developers to visualize and edit dynamically and easily software architecture, such as removing or updating existing components (change state, modify names...), adding new components to the architecture (services, interfaces,...). The output is a new software architecture.
   - Automatic Software Architecture Generation and Software Architecture As XMI File: The last step of the *Dynamic Software Architecture Evolution* process to generate automatically the new software architecture (with the XMI version) and the evolved system.

4.2.2 Proposed OSGi Meta-Model

Our proposed OSGi meta-model presents in Figure 4.2, it defines the architectural elements of OSGi, the dynamic component system for Java. Our meta-model based on an existing OSGi meta-model in the literature which is presented in [Ker+19].
In our meta-model, we added the dynamic aspect by adding the attribute *State* to PluginElement class. Also we added four new classes extend from Connector class (ExtensionConnector, PackageConnector, InterfaceConnector, and ServiceConnector), and more attributes to PluginElement class (*source*, *output*, *targetPlatform*, *withActivator*, *ulPlugin*, *rac*, *useTemplate*, and *expost*). The meta-model is available on Google Drive[5].

[5]Our meta-model: `https://drive.google.com/file/d/15k4xXs2Ur6MMYU7nr7yski7b2eYBjXYh/view?usp=sharing`

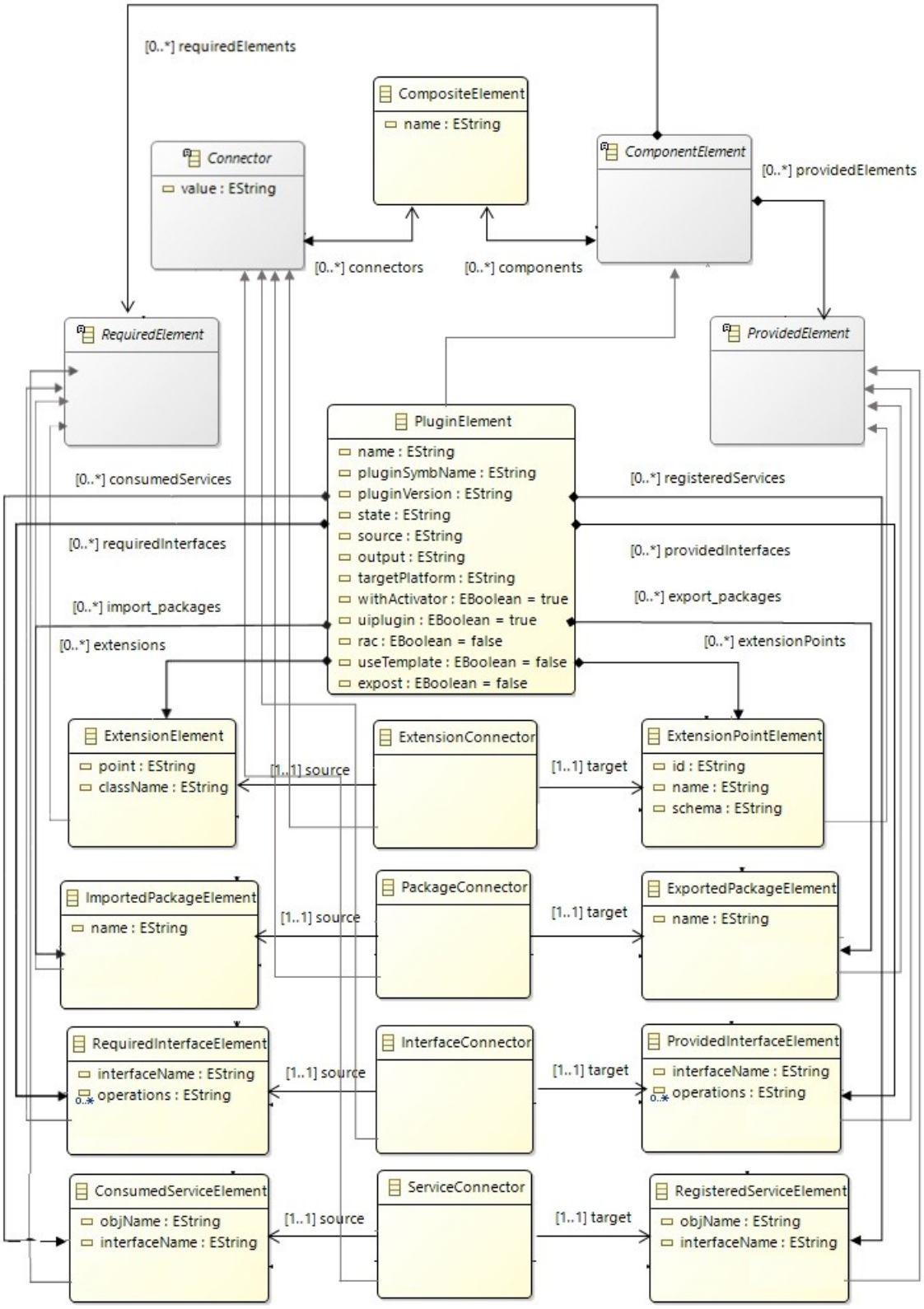

Figure 4.2: OSGi Meta-Model.

The OSGi meta-model is composed of a set of meta-classes, and they are:

- **CompositeElement:** The root element of our meta-model, it represents the container of all architecture components, it composed of many components and connectors.
- **ComponentElement:** A meta-class, it represents the component elements of the software architecture. It composed of several *ProvidedElement* and *RequiredElement*, and it is the base class of *PluginElement*.
- **Connector:** A meta-class, it represents a connection between the *ProvidedElement* and the *RequiredElement*. Also, it is the base class of *ExtensionConnector*, *PackageConnector*, *InterfaceConnector*, and *ServiceConnector*.
- **ProvidedElement:** A meta-class, it represents the provided elements of the component in the software architecture. It managed by *ComponentElement* and it is the base class of *ExtensionPointElement*, *ExportedPackageElement*, *ProvidedInterfaceElement*, and *RegisteredServiceElement*.
- **RequiredElement:** A meta-class, it represents the required elements of the component in the software architecture. It managed by *ComponentElement* and it is the base class of *ExtensionElement*, *ImportedPackageElement*, *RequiredInterfaceElement*, and *ConsumedServiceElement*.
- **PluginElement:** A meta-class, it represents the OSGi component (or bundle), it extends of *ComponentElement*. It composed of several extensions (*ExtensionPointElement* and *ExtensionElement*), packages (*ExportedPackageElement* and *ImportedPackageElement*), interfaces (*ProvidedInterfaceElement* and *RequiredInterfaceElement*), and services (*RegisteredServiceElement* and *ConsumedServiceElement*). Also, it has a set of attributes that are:

  1. *name:* is the name of the plugin (bundle).
  2. *pluginSymbName:* is the symbolic name of the plugin, specified by its Bundle-SymbolicName manifest header.
  3. *pluginVersion:* is the version of the plugin, specified by its Bundle-Version manifest header.
  4. *state:* is the state of the plugin, and it can be in **ACTIVE** state or **INSTALLED**, **RESOLVED**, **STARTING**, **UNINSTALLED**, or **STOPPING** state.

5. *source:* is the plugin source folder, its value is specified by build.properties and it is "src".
6. *output:* is the plugin output folder, its value is specified by build.properties and it is "bin".
7. *targetPlatform:* target platform show if the plugin is targeted to run with an OSGi framework or eclipse version.
8. *withActivator:* generate an activator (a java class that controls the plugin's life cycle) or no.
9. *uiplugin:* the plugin will make contributions to the UI (user interface) or no.
10. *rac:* create a rich client applications or no.
11. *useTemplate:* create a plugin using one of the templates.

- **ExtensionPointElement:** A meta-class, it represents the extension points elements of the plugin, define new function points for the platform that other plugins can plug into. It extends of *ProvidedElement* and managed by *PluginElement*, it has three attributes:

  1. *id:* is the identifier of the extension point.
  2. *name:* is the name space identifier of the extension point.
  3. *schema:* is the reference to the extension point schema.

- **ExtensionElement:** A meta-class, it represents the extensions elements of the plugin, the central mechanism for contributing behavior to the platform. It extends of *RequiredElement* and managed by *PluginElement*, it has two attributes:

  1. *point:* is the extension point to which this extension should be contributed.
  2. *className:* is the name of the generated java class/interface.

- **ExtensionConnector:** A meta-class, it represents a connection between an extension point element (target) and an extension element (source).

- **ExportedPackageElement:** A meta-class, it represents the exported packages elements, the packages that the plugin exposes to clients, specified by its Export-Package manifest header. The *name* attribute is the name of the exported package.

- **ImportedPackageElement:** A meta-class, it represents the imported package element, the packages on which this plugin depends without explicitly identifying their originating plugin, specified by its Import-Package manifest header. The *name* attribute is the name of the imported package.
- **PackageConnector:** A meta-class, it represents a connection between an exported package element (target) and an imported package element (source).
- **ProvidedInterfaceElement:** A meta-class, it represents the provided interfaces elements, the interfaces that this plugin provided to other plugins. It has two attributes:
  1. *interfaceName:* is the name of the provided interface.
  2. *operations:* is the list of the operations that this plugin provided from this interface.
- **RequiredInterfaceElement:** A meta-class, it represents the required interfaces elements, the interfaces that this plugin used from other plugins. It has two attributes:
  1. *interfaceName:* is the name of the required interface.
  2. *operations:* is the list of the operations that this plugin used from this interface.
- **InterfaceConnector:** A meta-class, it represents a connection between a provided interface element (target) and a required interface element (source).
- **RegisteredServiceElement:** A meta-class, it represents the registered services elements, the services that registered by this plugin. It has two attributes:
  1. *objName:* is the name of the service object registered by this plugin.
  2. *interfaceName:* is the name of the provided interface.
- **ConsumedServiceElement:** A meta-class, it represents the consumed services elements, the services that this plugin is using. It has two attributes:
  1. *objName:* is the name of the service object this plugin is using.
  2. *interfaceName:* is the name of the required interface.
- **ServiceConnector:** A meta-class, it represents a connection between a registered service element (target) and a consumed service element (source).

4.3 Software Architecture Recovering

The starting of the first step of the software architecture recovering process is during the execution of one of the use cases of the application, in which the "Use Case 0" is the launch of the application without applying a specific use case.

4.3.1 Static Analysis

Static Analysis is the first step of the software architecture recovering process, it aims to:

- Definite the entities of the system.
- Finding the relationship between system entities to get the necessary information to move to the next step, the recovering of software architecture at runtime (Dynamic Analysis).

4.3.2 Dynamic Analysis

The goal of this step is to recover the software architecture at runtime and that by identify the architectural elements of the system through "Bundles". After extracted the bundles of the software architecture, the role comes on the elements (components) associated with them. From the extracted bundles:

- Recovering the extension and the extension point.
- Recovering the imported and the exported packages.
- Recovering the provided and the required interfaces.
- Recovering the registered and the consumed services.

4.3.3 Component Architecture Generation

Component Architecture Generation is the last step in the software architecture recovering process. In this step first creating the XMI file version of the recovered software architecture at runtime by applying the following steps:

- Create the *CompositeElement*, the software architecture elements container.
- For each bundle of the software architecture, we create a *PluginElement* to represent it.

- For each bundle, we create their associated elements(*ExtensionElement* and *ExtensionPointElement*, *ImportedPackageElement* and *ExportedPackageElement*, *ProvidedInterfaceElement* and *RequiredInterfaceElement*, *RegisteredServiceElement* and *ConsumedServiceElement*).

After that, we get four XMI files for each architecture element type (extensions, interfaces, packages, and services) to make it easier. Now, generating an instance of our meta-model based on the previous XMI files of the recovered software architecture at runtime with the extension ".newcompo".
The generated component model is transformed into a graphical architecture to represent the recovered software architecture at runtime with its components and how they interact with each other in an abstract view.

4.4 Dynamic Software Architecture Evolution

4.4.1 Graphical Visualization & Comprehension of Software Architectures

The first step of the dynamic software architecture evolution process aims to represent the software architecture graphically with a high-level of abstraction to make its comprehension and the next steps of this process easier for the developers.

4.4.2 Update the Architecture

The graphical visualization allows developers to edit easily the recovered software architecture, so the goal of this step is to make a visual interaction between the developers and the software architecture easily and give them different options to update this architecture.

- Add New Components: In this case, we allow the developer to add new components to the software architecture, such as adding new plugins, new services, interfaces, and packages once right-click.
- Update Components: We give the possibility to developer to update different components, such as the ability to stop or start plugins and change components names.
- Remove Components: We allow to developer to remove any component from the software architecture (plugins, services, packages, extensions, or interfaces).

4.4.3 Automatic Software Architecture Generation & Software Architecture As XMI File

After updating and evolving the recovered software architecture, now we go to the last step of the dynamic software architecture evolution process to generate automatically the new software architecture (with the XMI file version) and the evolved system.

4.5 Conclusion

In this chapter, we introduced the process of our approach to the evolution of software architecture for component/service-oriented systems. Also, we introduced the OSGi meta-model and explained its elements. in the next chapter, we will present our implementation for our proposed approach process and the frameworks used.

Chapter 5

Implementation and Case Study

5.1 Introduction

In this chapter, we introduce the global and detailed architecture of our tool *ArchDynEvol*. We present also the implementation of our proposed approach process, the frameworks and tools we use, and the steps we take to create our application.
We provide in this chapter a case study based on the Eclipse (example) illustrating the use of our application.

5.2 *ArchDynEvol* Tool

In this section, we present the global and detailed architecture of our tool *ArchDynEvol*, and all the steps we are taking to develop and implement our application on Eclipse.

5.2.1 Architecture of *ArchDynEvol*

Figure 5.1 represents the global architecture of our tool. It is represented in UML Component Diagram using Visual Paradigm Online (VP Online) Express Edition. It composed of three main components: Software Architecture Recover, Graphical Software Architecture Editor, and Software Architecture Evolver.

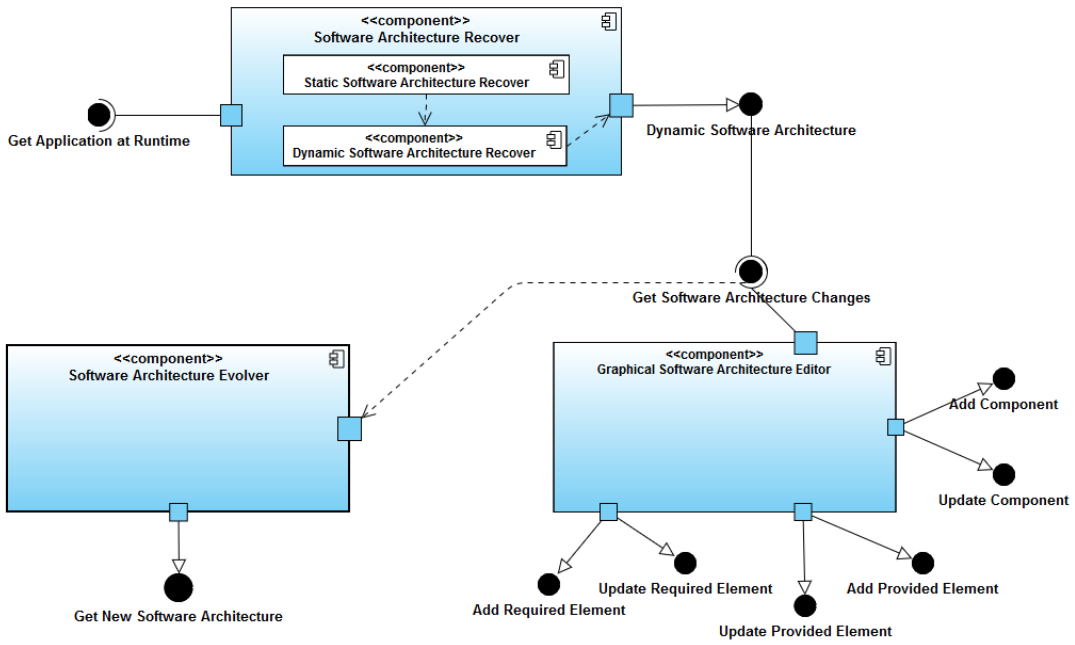

Figure 5.1: *ArchDynEvol* global Architecture.

1. **Component "*Software Architecture Recover*"**
   The component "*Software Architecture Recover*" recovers the software architecture from the application at runtime and generated the dynamic software architecture as result. It composed oh two sub-components:

   (a) **Component "*Static Software Architecture Recover*"**
   This component recovers the software architecture from the application, to identify the application entities and their relationships between them.

   (b) **Component "*Dynamic Software Architecture Recover*"**
   This component recovers the software architecture from the application at runtime, to extract their all components. These recovered components used to generate the dynamic software architecture.

2. **Component "*Graphical Software Architecture Editor*"**
   The component "*Graphical Software Architecture Editor*" represents the graphical editor that use to visualize graphically the recovered software architecture and to change it. These changes are presented as a set of operations add or update component, provided element, or required element.

3. **Component "*Software Architecture Evolver*"**
   The Component "*Software Architecture Evolver*" evolves the recovered software

architecture with the help of the component *Graphical Software Architecture Editor* by updating the architecture graphically to generate new software architecture.

5.2.2 Development of Graphical Software Architecture Editor

We present here all the steps that we take to develop our graphical software architecture editor.

5.2.2.1 OSGI EMF Meta-Model

The Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model [Gro].
EMF meta-model based on ecore file and genmodel file. In this step, we present the way for creating EMF meta-model files.

Ecore Model Creation

Ecore meta-model generate the entity classes of our application, to create it we follow these steps:

- First, we create an empty EMF project by clicking "File" → "New" → "Other...", choose from Eclipse Modeling Framework "Empty EMF Project", then clicking at "Next" and giving a name to this project then "Finish" as Figure 5.2 show.

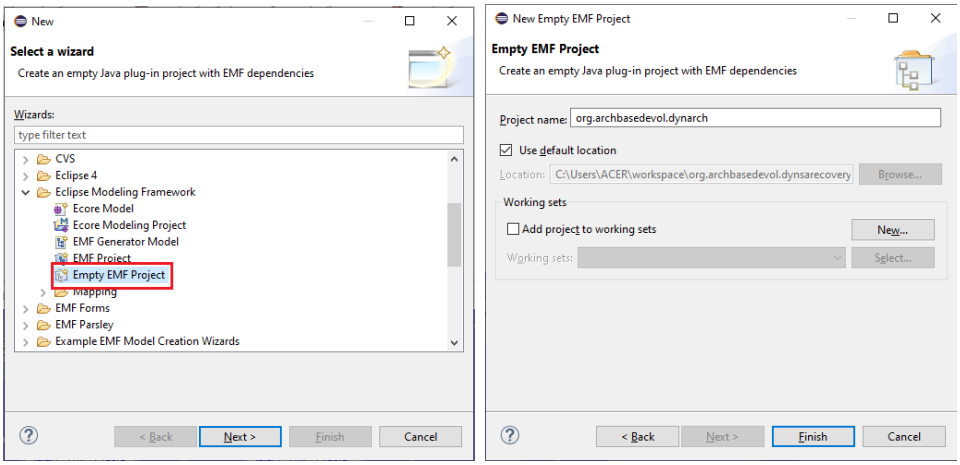

Figure 5.2: Empty EMF Project Creation Steps.

- Now, we create the Ecore file. First, we go to the "model" folder on the created project, right-clicking "New" → "Other" and choosing from Eclipse Modeling Framework "Ecore Model", clicking "Next" then giving a name and clicking "Finish" button as the Figure 5.3 show.

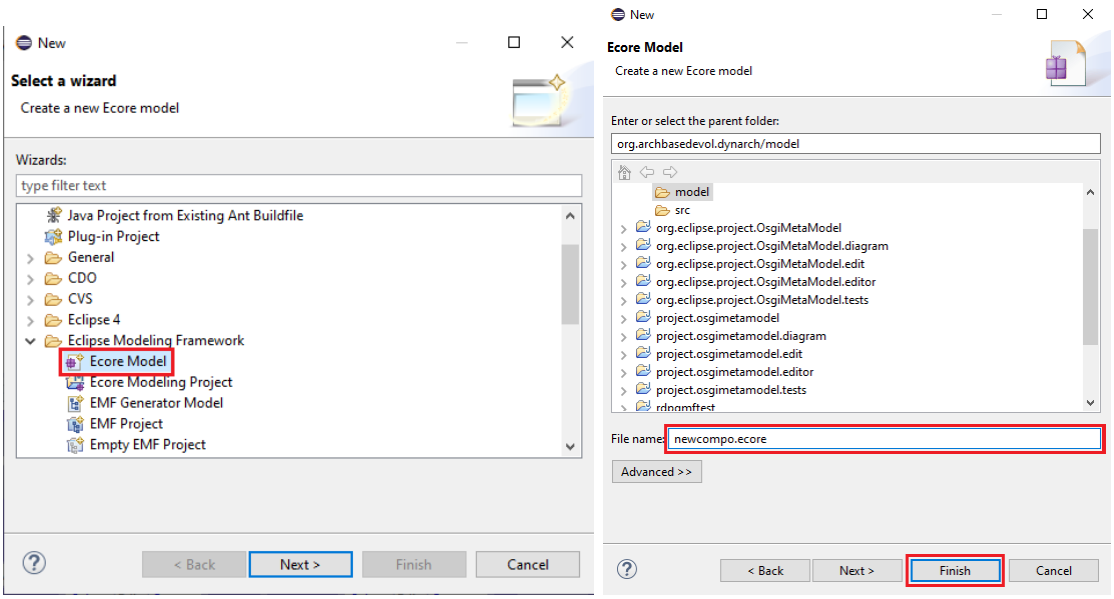

Figure 5.3: Ecore Meta-Model Creation Steps.

- In the properties view, we give the package of the new model a name and a URI as shown in Figure 5.4. We named the package and Ns Prefix the same "newcompo", and change Ns URI to "platform:/resource/org.archbasedevol.dynarch/model/newcompo.ecore".

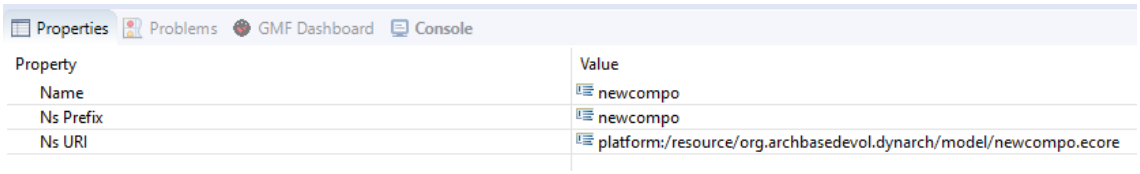

Figure 5.4: The Properties View.

- Now, the addition of the Ecore model meta-classes (CompositeElement, PlugInElement, Connector,...), and they are of type EClass. Our final Ecore model "newcompo.ecore" is shown in Figure 5.5.

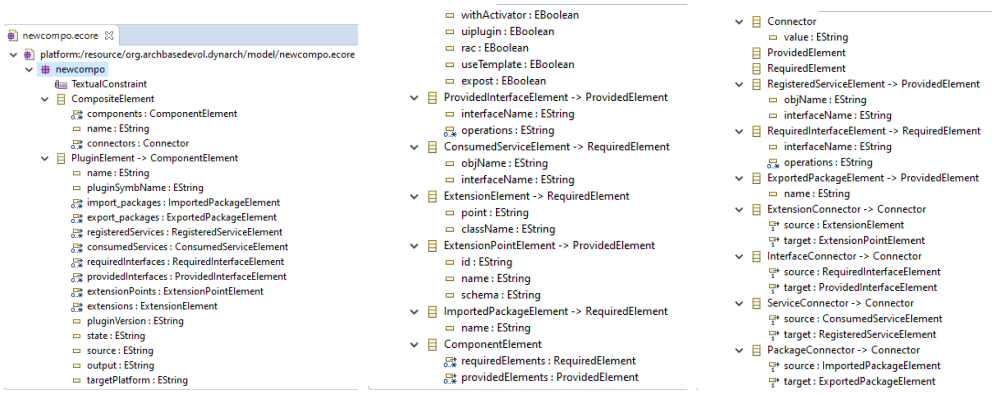

Figure 5.5: The Ecore model "newcompo.ecore".

Genmodel Creation

Genmodel file contains information for the generation of the Ecore model entities and code, to create it we follow these steps:

- First, we show "GMF Dashboard" view by clicking at "Window" → "Show View" → "other..." in "general" folder choose "GMF Dashboard", such as figure 5.6 showing.

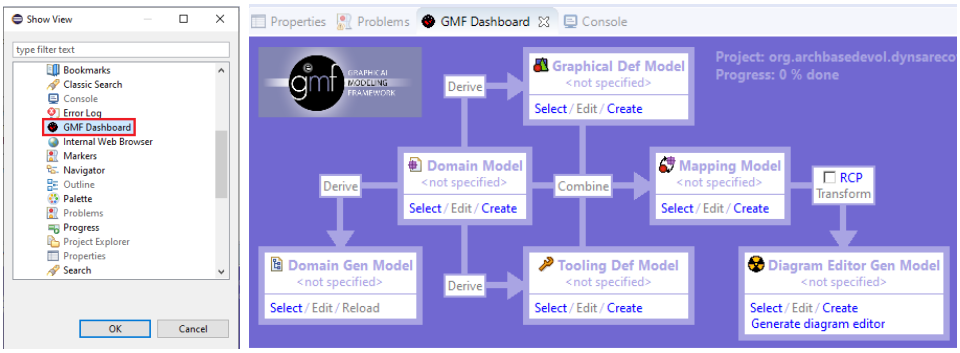

Figure 5.6: GMF Dashboard View.

- From the GMF Dashboard view, we click on "Select" of "Domain Model" then choosing the created Ecore model → "Drive". We give a name to this genmodel "newcompo.genmodel", clicking "Next" → Ecore model → "Next" → "Load" → "Finish". (see Figure 5.7)

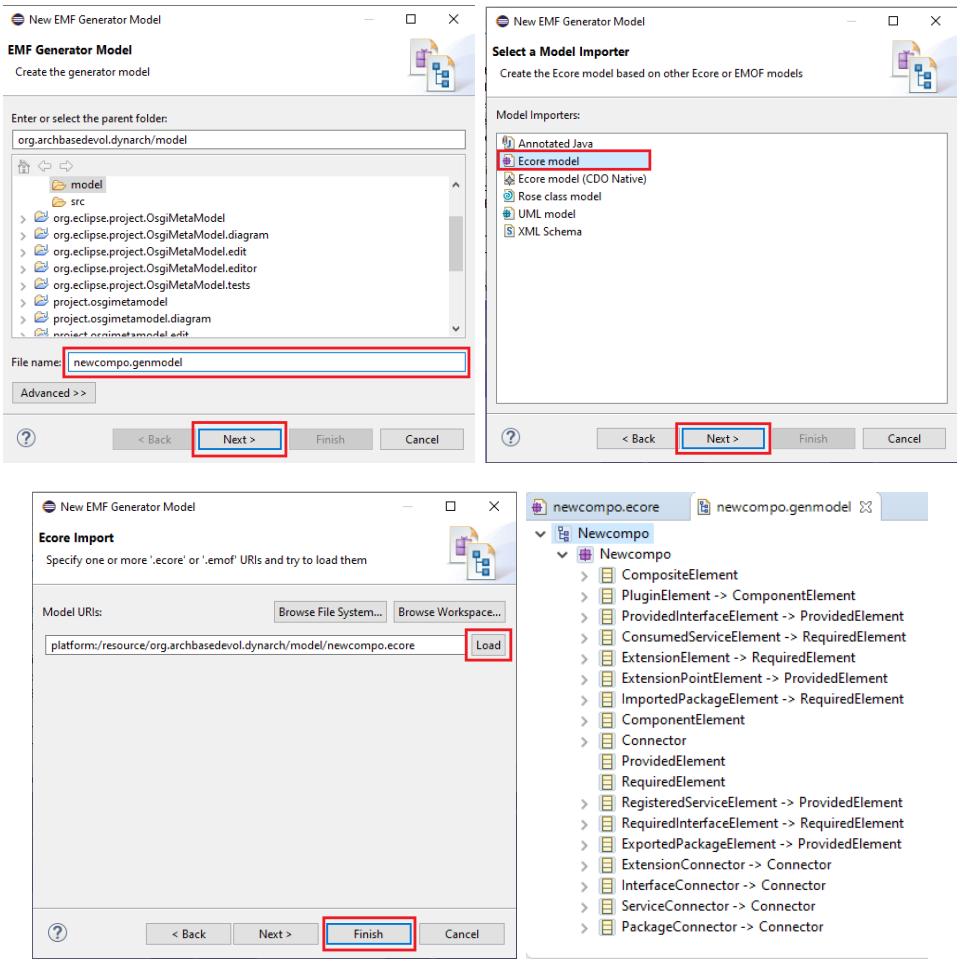

Figure 5.7: "newcompo.genmodel" Creation Steps.

- Then, we go to the root node of the "newcompo.genmodel" to set the compliance level to 6.0 from the properties view.

We right-click to the root node of genmodel and choose "Generate All", that will give three packages in the project "src" folder which contains entities that help the creation of the model instances, and other three plugins "edit", "editor", and "tests" as Figure 5.8 show.

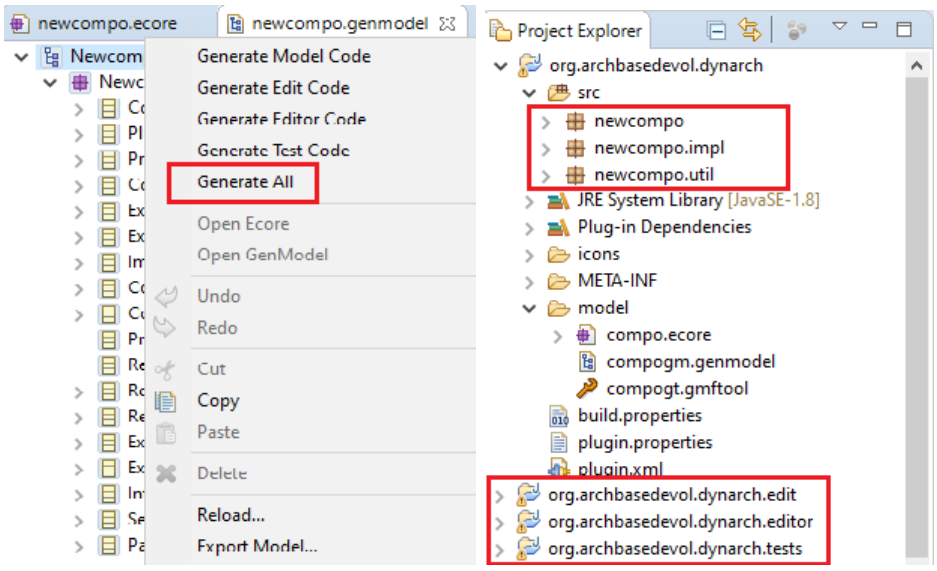

Figure 5.8: Source Code Generation Steps.

5.2.2.2 Graphic Editor Creation

The Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF) [Wik].
We use the graphic editor to visualize the recovered software architecture at runtime and to make the evolution task easier for developers. To create it we follow these steps:

- Tooling definition model creation.
- Graphical definition model creation.
- Mapping model creation.
- Diagram editor gen model creation.

Tooling Definition Model Creation

This model presents the palette and menu elements of the graphic editor. To create the tooling definition model first we click "Drive" of "Tooling Def Model" on the dashboard view, then we give "newcompo.gmftool" as a name → "Next" → "Load" → we select the root, in our case is "CompositeElement" → "Next" → then checking the palette elements → "Finish". (see Figure 5.9)

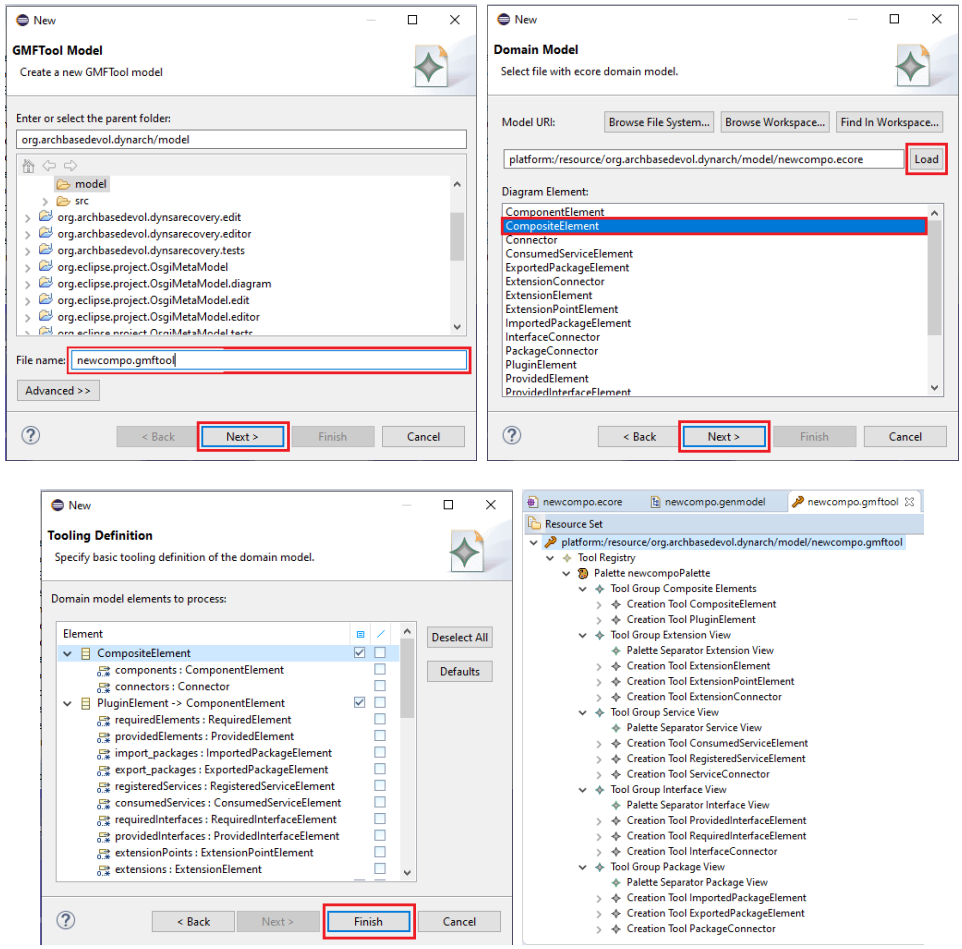

Figure 5.9: Tooling Definition Model Creation Steps.

We can change the palette icons by placing the new ones in folder "icons" in the "edit" plugin (edit → icons → full → obj16), and it should have the same name and with "GIF" extension.

Figure 5.10 shows our palette and menu elements of the graphic editor.

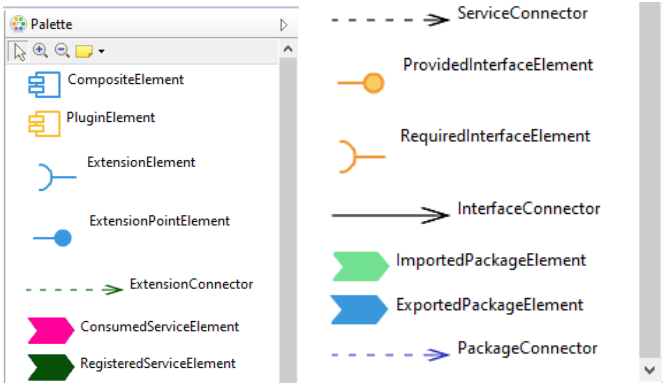

Figure 5.10: Graphical Definition Model Creation.

Graphical Definition Model Creation

This model defines the graphical editor's surface elements. To create the graphical definition model first we click "Drive" of "Graphical Def Model" on the dashboard view, then we give "newcompo.gmfgraph" as a name → "Next" → "Load" → we select the root (CompositeElement) → "Next" → now checking our elements that must viewed on the editor surface → "Finish", as Figure 5.11 show.

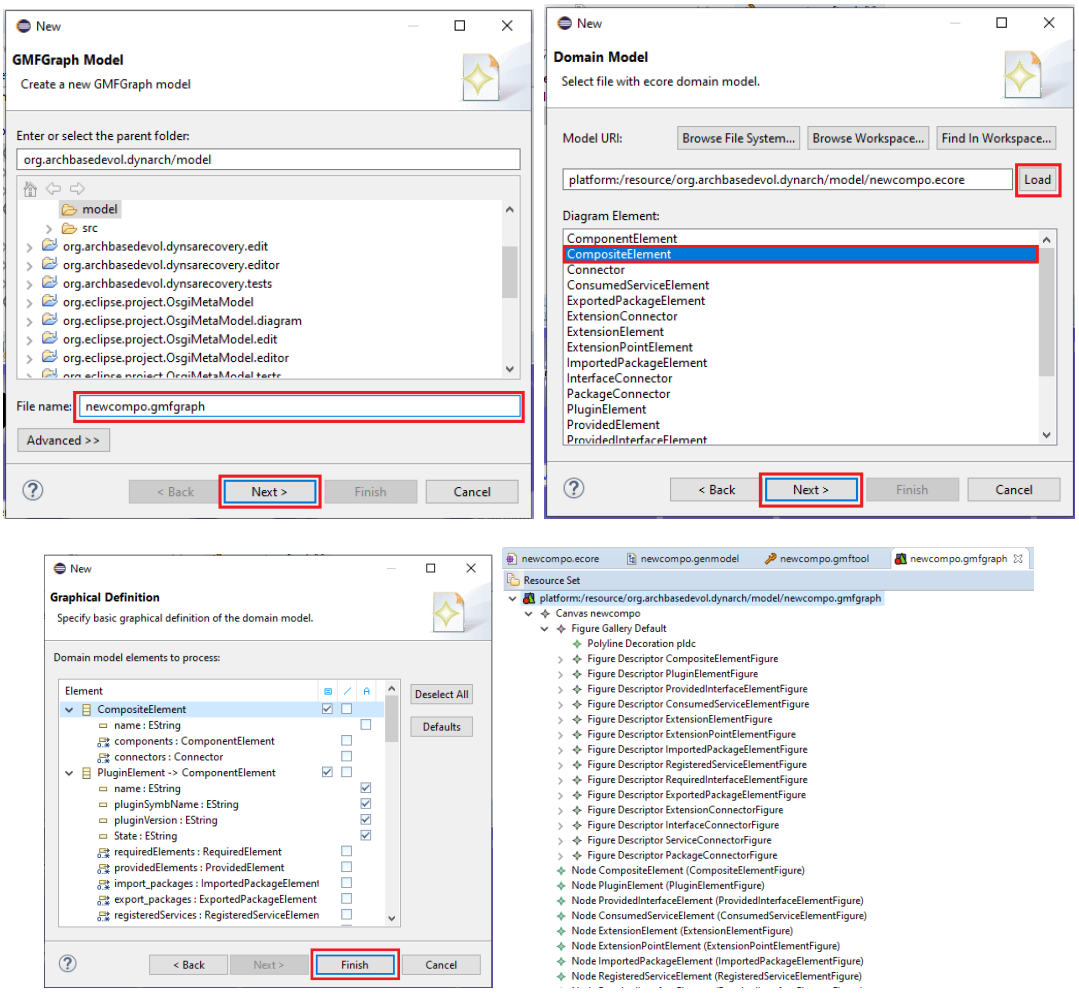

Figure 5.11: Graphical Definition Model Creation.

We can change the graphical editor's surface default elements icons such as setting its size, color, and layout... by the graphical def model. (see Figure 5.12)

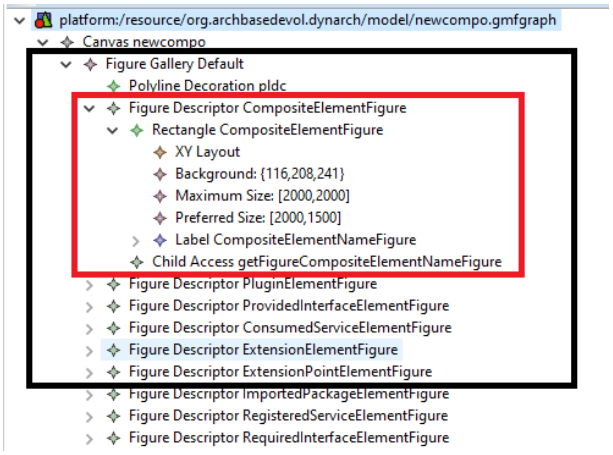

Figure 5.12: Changement of Elements Figure Descriptor.

Also we can replace the old icons with new ones have "svg" extension, and that by:

1. Download and add the "org.eclipse.gmf.runtime.lite.svg" jar in the Eclipse plugin.
2. Go to "Figure Descriptor" of the element → right-click → New Sibling → SVG Figure.
3. Go to the properties view, give a name to the SVG figure, and set the document URI where the SVG icons will be in this project ("diagram" plugin → icons → figure), as shown Figure 5.13.

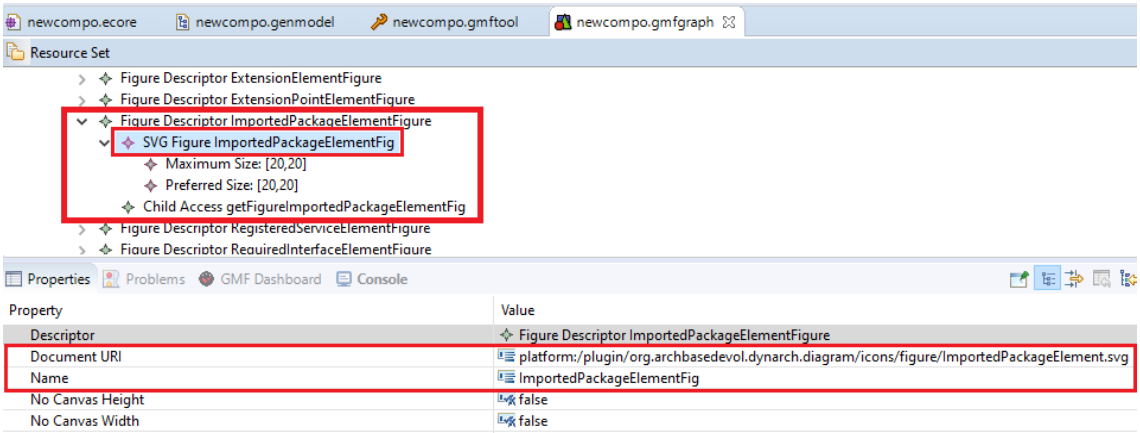

Figure 5.13: SVG Figure Creation.

Mapping Model Creation

This model binds the three previous models: the domain, the graphical definition, and the tooling definition. To create the mapping model first we click "Combine" in the

dashboard, then giving "newcompo.gmfmap" as a name → "Next" → "Load" → we select the root element (CompositeElement) → "Next" → "Load" → "Next" → "Load" → "Next". (see Figure 5.14)

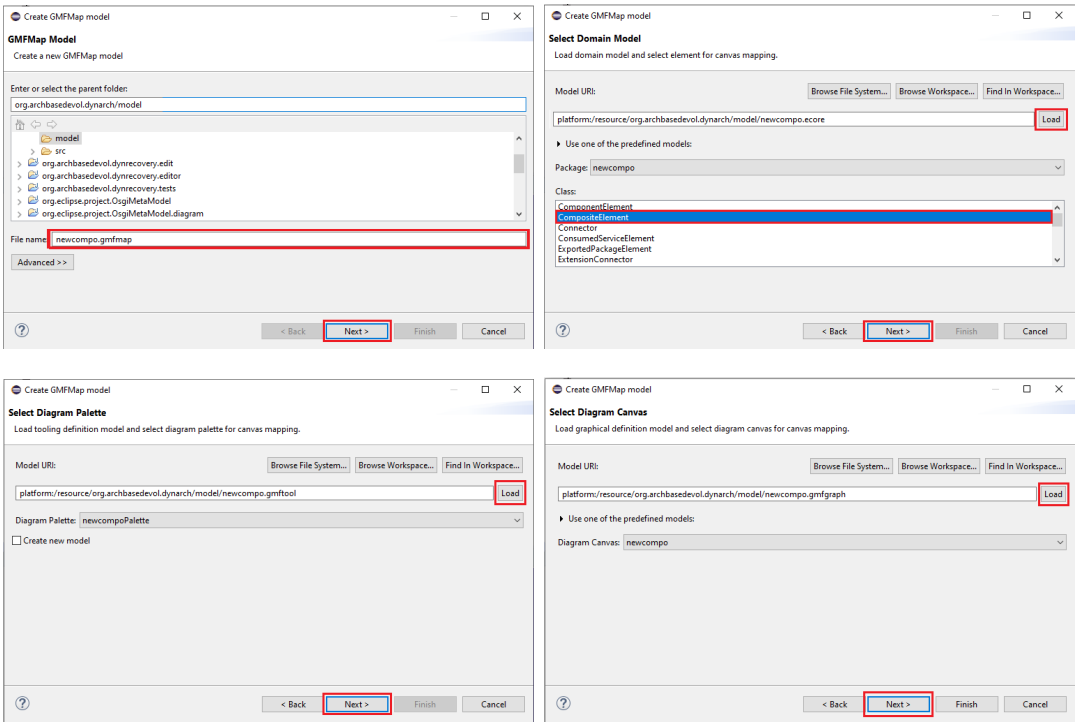

Figure 5.14: Mapping Model Creation Steps.

In the map domain model elements, we find two sections (Nodes and Links) and they represent:

- In the "Nodes" section, we must select only the elements that correspond with our architecture nodes (PluginElement, ImportedPackageElement, ExportedPackageElement...).
- In the "Links" section, we must select only the elements that correspond with our architecture links (InterfaceConnector, PackageConnector, ExtensionConnector, and ServiceConnector).

Now for each link, it is important to check its properties and that by, select the link → "Change..." → and verify the values of "Source Feature" and "Target Feature", "Diagram Link", and "Tool" → "OK". (see Figure 5.15)

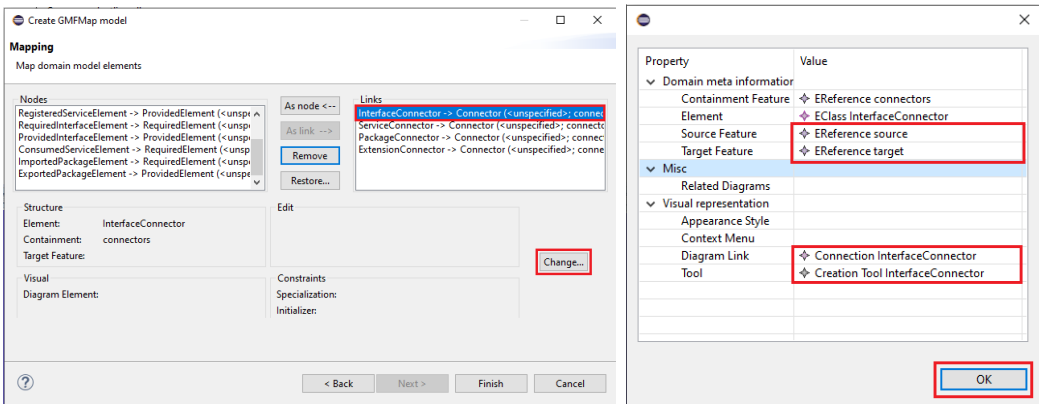

Figure 5.15: Map Domain Model Elements.

After the generation of our mapping model, we must verify the "Tool" and "Diagram Node" properties for each "Node Mapping". Also, we must verify the "Containment Feature" and "Reference Child" properties for each "Child Reference" of our mapping model, as Figure 5.16 shown.

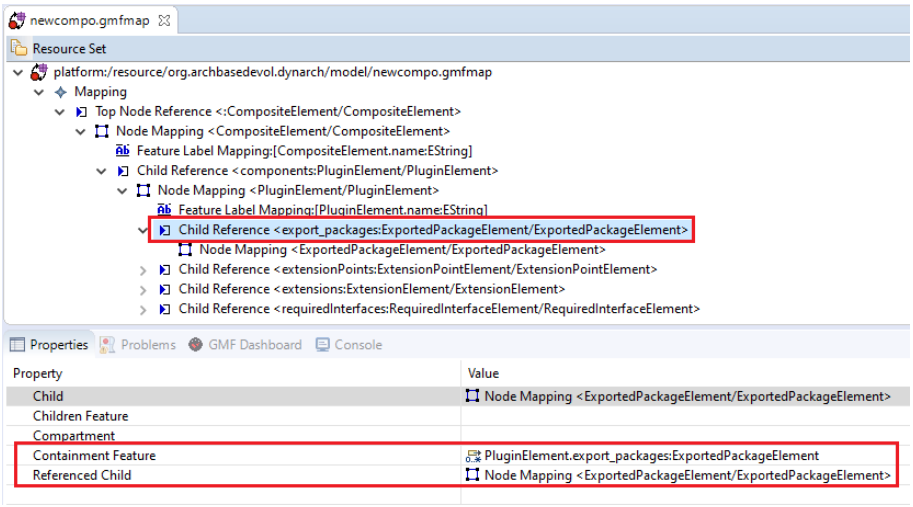

Figure 5.16: Our Mapping Model "newcompo.gmfmap".

Diagram Editor Gen Model Creation

This is the last step in the graphic editor creation. First, we go to the dashboard and check the "RCP" box → clicking "Transform", that will give the model "newcompo.gmfgen" → clicking "Gen Editor Generation" and configure its properties. We return to the dashboard and clicking on "Generate Diagram Editor" to generate the plugin "diagram". (see Figure 5.17)

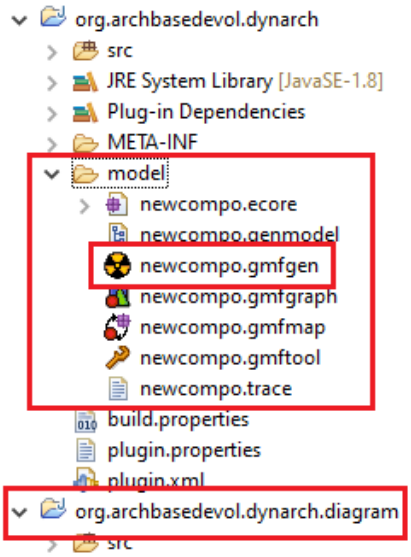

Figure 5.17: The "diagram" plugin and the model "newcompo.gmfgen".

Note: After the generation of the plugin "diagram", add a new folder named "icons" (for the SVG icons of the graphical definition model creation step).

5.2.3 Recovering Software Architecture Components

The Eclipse platform relies on plugins which are represented as bundles in OSGi specification. So we can represent our architecture via OSGi bundles, for that we use classes in *"org.osgi.framework"* package to get the bundles at runtime, such as class *"Bundle"*, *"BundleContext"*, and *"FrameworkUtil"* class. For each recovered bundle, we identify its name, symbolic name, version, state, source, target, output, and its target platform (location). These specifications are different in each bundle.
Then, we can extract the rest components from these bundles.

- **Consumed and Registered Services:** We can recover both kinds of services of a bundle using the methods in the *"Bundle"* class from *"org.osgi.framework"* package. In which, we use the method *".getServicesInUse()"* for consumed services and *".getRegisteredServices()"* for registered service.

- **Required and Provided Interfaces:** We can recover the bundle interfaces from its services. In which, the required interfaces are related by the consumed services and the provided interfaces by the registered services.

- **Imported and Exported Packages:** We can recover the bundle packages from its MANIFEST.MF file, for that we developed a class *"ManifestParser.java"* to parse bundle manifest file and reset its import and export packages.

- **Extensions and Extension Points:** To recover the bundle extensions and extension points, we need to use another package "org.eclipse.core.runtime". For the extensions we use the method ".getExtensions()" from "IExtensionPoint" class and for the extension points the method ".getExtensionPoints()" from "IExtensionRegistry" class.

5.2.4 Implementation of Evolution Actions

Before starting the evolution task for the system, we need to create the XMI version (instance) of our recovered software architecture at runtime to use it in the graphical presentation. To create it we follow these steps:

- Initialization of the model then retrieving the default factory singleton.
- Creation of the content of the model:
  1. We create first the "CompositeElement" that represents the recovered software architecture elements container at runtime.
  2. Now, we create a "PluginElement" for each bundle of the software architecture and its all attributes (Name, State, Version...).
  3. Now for each "PluginElement" that we created, we create its associated elements (ExtensionElement and ExtensionPointElement, ImportedPackageElement and ExportedPackageElement, ProvidedInterfaceElement and RequiredInterfaceElement, RegisteredServiceElement and ConsumedServiceElement) also with all their attributes.
- Saving the XMI file.

Figure 5.18 presents a part of the XMI file of recovered software architecture at runtime with service elements.

```
<?xml version="1.0" encoding="ASCII"?>
<newcompo:CompositeElement xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:newcompo="platform:/resource
  <components xsi:type="newcompo:PluginElement" name="Plugin 0" pluginSymbName="org.eclipse.osgi" pluginVersion="3.11.3.v20170209-1843" state="ACTIVE" source="src" outp
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@21cb2a8f" interfaceName="org.osgi.service.log.LogReaderService"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@24637680" interfaceName="org.osgi.service.log.LogService"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@5b1513" interfaceName="org.eclipse.osgi.framework.log.FrameworkLog"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@2d641670" interfaceName="org.eclipse.osgi.service.datalocation.Location"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@5583de01" interfaceName="org.eclipse.osgi.service.environment.EnvironmentI
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@6aeb7547" interfaceName="org.osgi.service.packageadmin.PackageAdmin"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@32401684" interfaceName="org.osgi.service.startlevel.StartLevel"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@3f9a412a" interfaceName="org.osgi.service.permissionadmin.PermissionAdmin"
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@74a2f30e" interfaceName="org.osgi.service.condpermadmin.ConditionalPermiss
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@4552e190" interfaceName="org.osgi.service.resolver.Resolver"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@77142b88" interfaceName="org.eclipse.osgi.service.debug.DebugOptions"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@1aeab751" interfaceName="java.lang.ClassLoader"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@79331d81" interfaceName="org.eclipse.osgi.service.urlconversion.URLConvert
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@2bbf2d56" interfaceName="org.eclipse.osgi.service.localization.BundleLocal
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@7da11bc2" interfaceName="javax.xml.parsers.SAXParserFactory"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@323e6b18" interfaceName="javax.xml.parsers.DocumentBuilderFactory"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@5300a7cb" interfaceName="org.eclipse.osgi.service.security.TrustEngine"/>
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@187f560" interfaceName="org.eclipse.osgi.signedcontent.SignedContentFactor
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@740d9199" interfaceName="org.eclipse.osgi.service.resolver.PlatformAdmin"/
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@35aa52ab" interfaceName="org.eclipse.osgi.service.debug.DebugOptionsLister
    <registeredServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@6342852a" interfaceName="org.eclipse.osgi.service.runnable.StartupMonitor"
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@5dd050d7" interfaceName="org.osgi.service.url.URLStreamHandlerService"/>
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@454993ad" interfaceName="org.eclipse.osgi.service.debug.DebugOptionsListener
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@2291bea5" interfaceName="org.eclipse.osgi.service.environment.EnvironmentInf
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@558d0408" interfaceName="org.eclipse.osgi.service.debug.DebugOptionsListener
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@a84b086" interfaceName="org.eclipse.osgi.framework.log.FrameworkLog"/>
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@3d99e80" interfaceName="org.eclipse.osgi.signedcontent.SignedContentFactory"
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@2c127cc8" interfaceName="org.eclipse.osgi.service.debug.DebugOptionsListener
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@11829006" interfaceName="org.eclipse.osgi.service.datalocation.Location"/>
    <consumedServices objName="org.eclipse.osgi.internal.serviceregistry.ServiceObjectsImpl@3ead1230" interfaceName="org.eclipse.osgi.service.runnable.StartupMonitor"/>
  </components>
```


Figure 5.18: XMI File of Recovered Software Architecture at Runtime.

These steps are executed during the software architecture at runtime recovered. Our tool generates an XMI file for each type of the architecture elements (that's mean four files). After the recovered of the software architecture at runtime and generate its XMI version, now to present this architecture graphically do this:

- Copying the XMI file in the "Project Explorer" view with changing its extension to ".newcompo".
- Clicking "File → "Initialize newcompo-diagram diagram file" → selecting the XMI file that we want evolving → "Finish".

Now, we can make our recovered software evolvable architecture through these operations:

- Changing the plugins state (Start or Stop).
- Adding new components to the recovered software architecture plugins (extensions and extension points, imported packages and exported packages, provided Interfaces and required interfaces, registered services and consumed services).
- Adding new plugins to software architecture.
- Changing the properties of components.
- Removing components from software architecture.

Changing The Plugins State

To make our software architecture plugins changing its state we follow the next steps:

- The first step is to create a menu that contains the actions, and that by opening the "MANIFEST.MF" file "META-INF" document in the project → clicking "Extensions" → "Add" → selecting "org.eclipse.ui.popupMenus" as an extension point filter → "Finish" → going to "Extension Details" and giving an ID to this extension. In the plugin define extension section, right-clicking the created popup menu extension → "New" → "objectContribution".

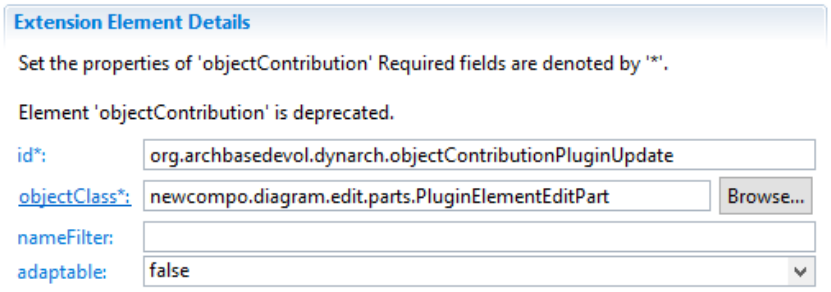

Figure 5.19: The "objectContribution" Properties Section.

- Now, we go to the "Extension Elements Details" to set the created objectContribution properties (See figure 5.19). Then, right-clicking the created objectContribution → "New" → "menu" → setting the menu properties, then right-clicking the menu → "New" → "separator".
  Note: We find the object class (PluginElementEditPart) in ".diagram/edit/parts".

- Then we create the state actions (Stop and Start), by returning to the objectContribution right-click → "New" → "action" → setting the action properties in the "Extension Elements Details" section. We make this step twice, for the Stop action and Start action. (see Figure 5.20)

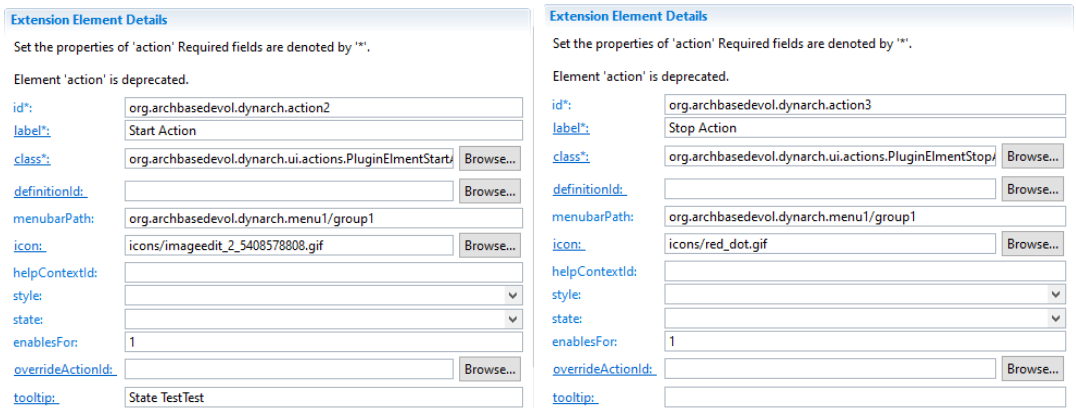

Figure 5.20: The "Start Action" and "Stop Action" Actions Properties Section.

- Now, we create the management class of the changement state action. It extends from "ActionDelegate" and implements from "IObjectActionDelegate". In this class, we find two principal methods, the first one determines the type of elements that we want to select (for us is the pluginElement). In the second method, we determine the type of action of the selected elements, we take "Stop Action" as an example. We test first the state of the selected plugin, if it is in "RESOLVED" or "STOPPING" state we give a warning message that it is impossible to make this changement, else this action is successful. With the setting of the plugin state to "STOPPING," there is removed to all the components elements that relate with the updated plugin.

Adding New Components: Services

In this step, we develop the possibility of adding new components to the software architecture, and we take "Services" as an example of components by following the next steps:

- First, we create a new menu for the addition of the services so we return to the created objectContribution right-clicking → "New" → "menu" → setting the menu properties, then right-clicking the menu → "New" → "separator".

- Then we create the addition services actions (Consumed and Registered), by back to the objectContribution right-click → "New" → "action" → setting the action properties in the "Extension Elements Details" section. We make this step twice, for the consumed services addition and the registered services addition.

- Now, we create the management class of the service addition action. It also extends from "Action-Delegate" and implements from "IObjectActionDelegate" such as the previous class (class of the state change) with the two principal methods, the first one responsible for the element selection and the second for the addition of services. First, we get all the system existed services and show it as a list of checkboxes, then we choose what services we want → OK, and the services are added at the selected plugin.

We take the same steps for the other operations such as adding new plugins to the architecture or remove it and changing components properties.

5.3 Case Study: Eclipse Based Applications

In this section, we present an example of running our application on the Eclipse platform.

1. To run our application we need first to create a new configuration. In "Run Configurations", we right-click "Eclipse Application" → "New" → setting the configuration name → "Apply" → "Run" (see Figure 5.21). After we click "Run", "Validation" is shown we select "org.apache.xmlrpc" → "OK".

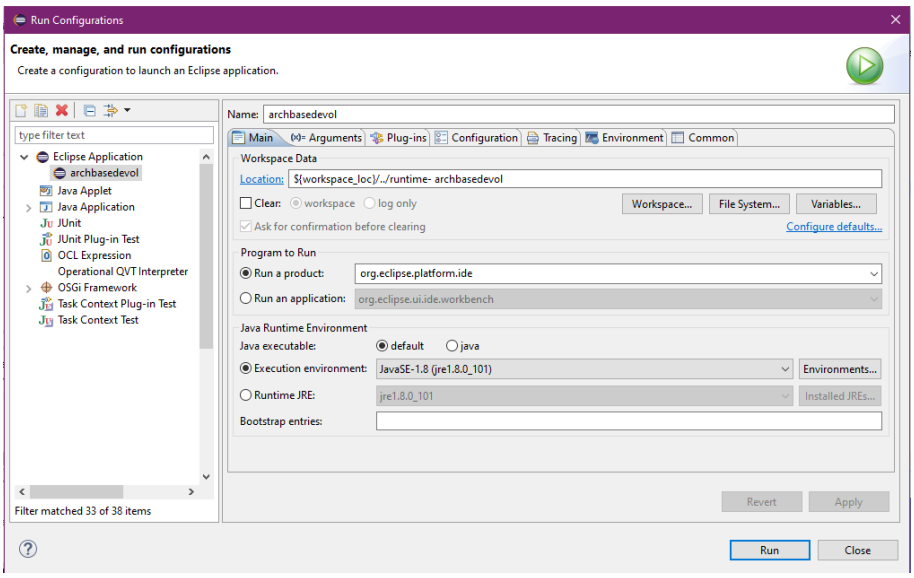

Figure 5.21: The Run Configuration of Our Application.

Because it is the first time to run this configuration we need to activate our plugin through the "Host OSGi Console". (see Figure 5.22)

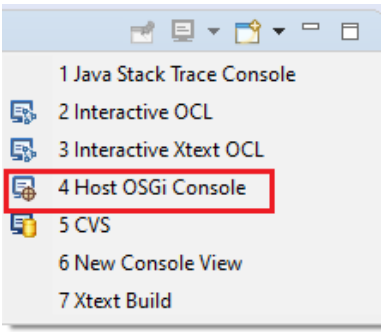

Figure 5.22: The Host OSGi Console.

2. Now we need to know the "id" of our plugin to activate it and that by typing "ss name of the plugin" in the Host OSGi Console (see Figure 5.23). After getting the id (at this moment our id = 59), now we activate our plugin by typing "start 59".

```
osgi> ss archbas
"Framework is launched."
id      State       Bundle
59      STARTING    org.archbasedevol.dynarch_0.1.0.qualifier
60      STARTING    org.archbasedevol.dynarch.diagram_1.0.0.qualifier
61      STARTING    org.archbasedevol.dynarch.edit_1.0.0.qualifier
62      STARTING    org.archbasedevol.dynarch.editor_1.0.0.qualifier
63      STARTING    org.archbasedevol.dynarch.tests_1.0.0.qualifier
```

Figure 5.23: Search Result for Plugin ID.

Note: This step is not necessary after the first run, we can do it just to make sure that our plugin is active.

With activating our plugin, the task of recovering software architecture also begins. We save the recovered software architecture in four XMI files (for services, extensions, interfaces, and packages) such as shown in Figure 5.18.

3. Then, we create a project document ("File" → "New" → selecting "General" → "Project" → giving it a name → "Finish") to put the recovered software architecture XMI file on it. Now, we copy the generated software architecture XMI file in the created project, in this example we take the services architecture file and changing its extension to ".newcompo".

4. To visualize our recovered software architecture graphically we follow these steps, clicking "File" → "Initialize newcomp-diagram diagram file" → selecting the file that coped → "Next" → selecting the diagram root (for us it is "CompositeElement") → "Finish". After that our software architecture is presented graphically as shows Figure 5.24.

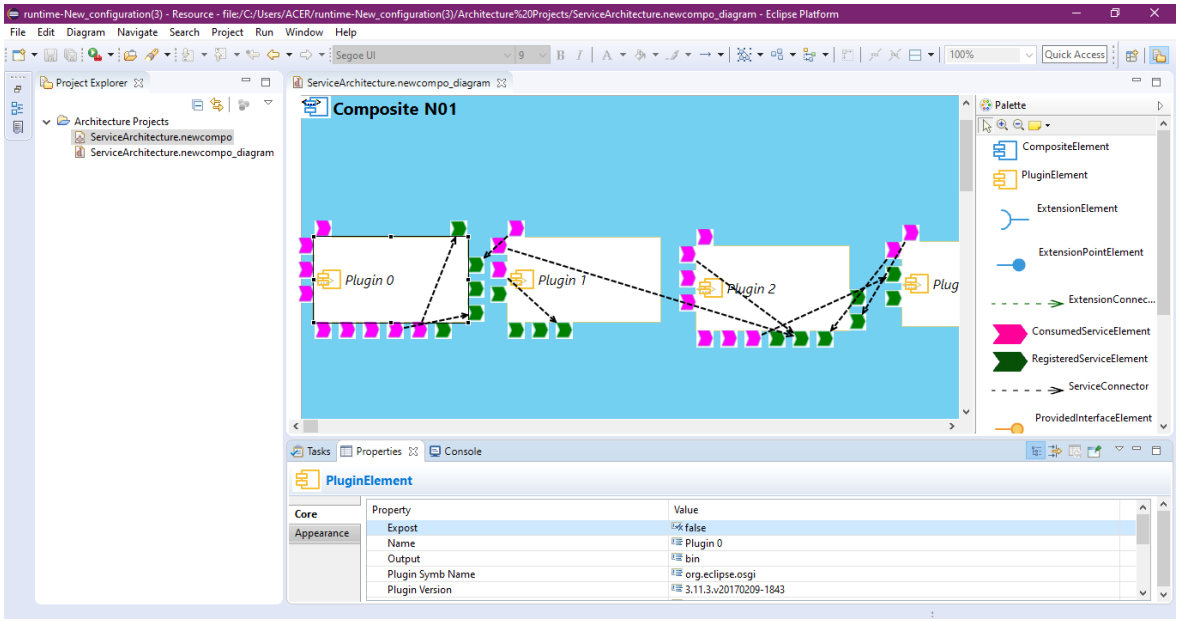

Figure 5.24: The Recovered Software Architecture in Our Graphic Editor.

Our Graphic Editor is composed of four parts:

- The Project Explorer view is where we put the recovered software architecture files and its diagrams.
- The space used to visualize and update the recovered software architecture.
- The palette and menu elements of the graphic editor provide the software architecture components. All our palette elements were shown in Figure 5.10.
- The properties view which provides the information of each component in the software architecture.

The composite (cyan surface) is the container for all components of the recovered software architecture and the plugin elements are represented with the white rectangles.

5. Now we can apply our methods of evolution (changing state, adding or removing components).

   - To changing the plugin state, we just select a plugin, right-clicking → "Runtime Change State" → and choosing one of the two options (see Figure 5.25).

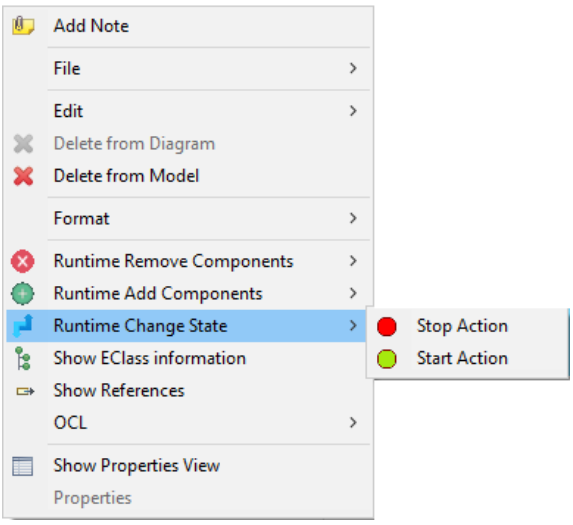

Figure 5.25: Runtime Change State Menu and Its Options.

If we choose "Stop Action" and the plugin state is "ACTIVE" or "STARTING", can note that its state in the properties view is changed to "STOPPING" and all the related components with this plugin are removed. But if the selected plugin state is "RESOLVED" or "STOPPING", a warning message is shown as Figure 5.26 shown.

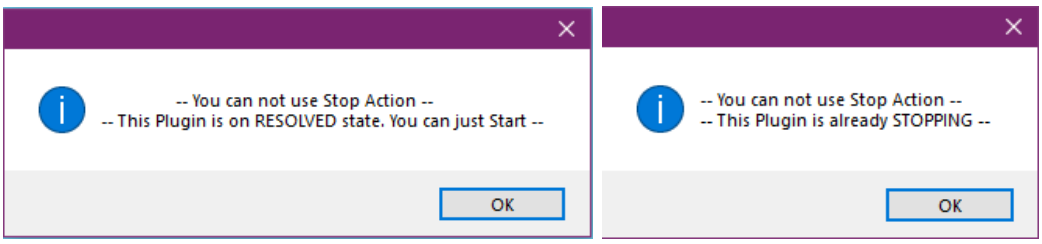

Figure 5.26: "RESOLVED" and "STOPPING" Warning Messages.

If we choose "Start Action" and the plugin state is "RESOLVED", "STARTING", or "STOPPING" can note that its state in the properties view is changed to "ACTIVE". But if the selected plugin state is "ACTIVE", a warning message. (see Figure 5.27)

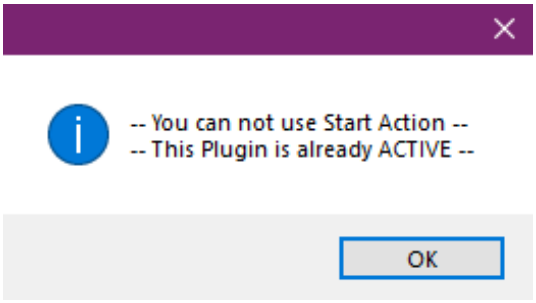

Figure 5.27: "ACTIVE" Warning Messages.

- To add services to a plugin (we take services as an example of components), we just select the plugin, right-clicking → "Runtime Add Components" → and choosing one of the two kinds of services. (see Figure 5.28)

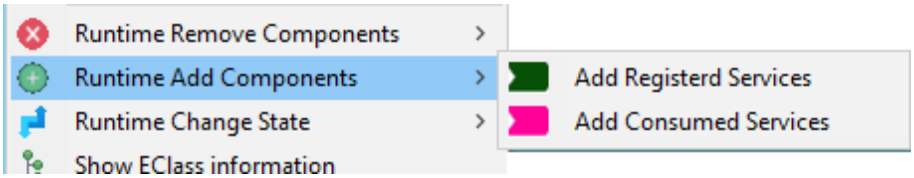

Figure 5.28: Runtime Add Components Menu (Services Options).

When choosing any of the addition of services options, a list of all the system services is shown (see Figure 5.29), then check the box of any service it will be added in the plugin → "OK".

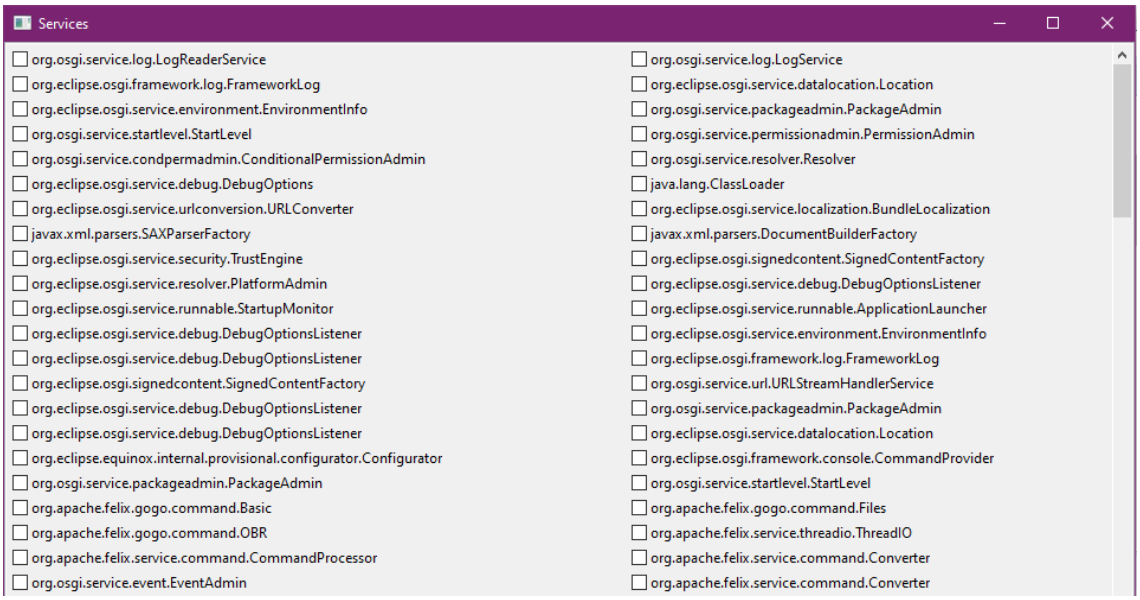

Figure 5.29: The System Services List.

- To remove services from a plugin (we take services as an example of components), we just select the plugin, right-clicking → "Runtime Remove Components" → and choosing one of the three options. (see Figure 5.30)

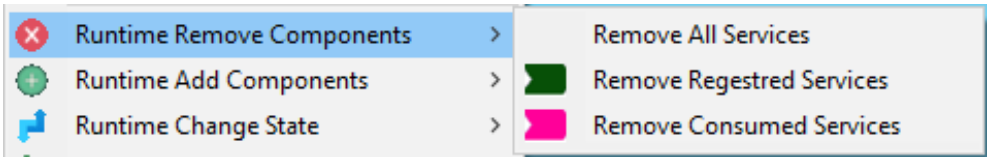

Figure 5.30: Runtime Remove Components Menu (Services Options).

If we choose "Remove Registered Services", all the registered services that related to the selected plugin are removed. The same action will happen with the consumed services if we choose "Remove Consumed Services". And if we choose "Remove All Services", both kinds are removing from the plugin.

- To add a new plugin to the architecture we select the composite, right-clicking → "Add Plugin". (see Figure 5.31)

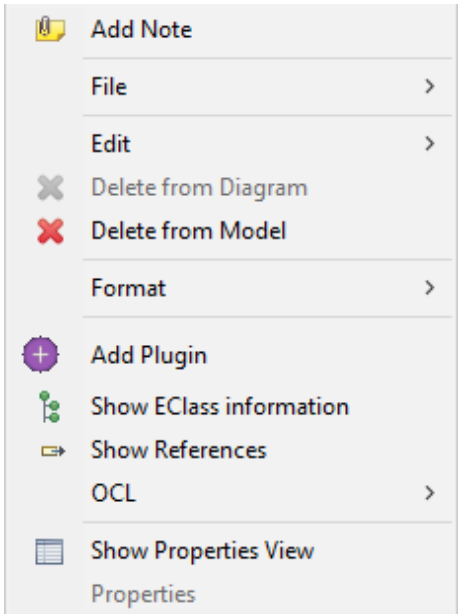

Figure 5.31: Add Plugin Action.

5.4 Conclusion

In this chapter, we introduced the global and detailed architecture of our tool *Arch-DynEvol* that we represented as a UML component diagram. We presented also the development tools and frameworks that we used to implement our proposed approach process. Then, we showed the steps that we took to create and develop our application. In the final, we provided an execution example of our application based on Eclipse.

General Conclusion

The dynamic evolution of software architecture means that the architecture can be modified or changed during runtime such as adding, removing, and updating its components or connectors. The goal of software architecture dynamic evolution is to reduce the time, effort, and complexity involved in the maintenance and evolution task of large software systems. In this project, our aim was to assist developers to evolve and maintain their systems during runtime via dynamic software architecture. We proposed a process divided into two sub-processes. *Software Architecture Recovering* is the first process, its aim recovering component-based architecture from applications at runtime. It starts with identifying the application entities and their relationship to each other from source code using static analysis. Then, recovering the application bundles and their associated elements at runtime from source code using dynamic analysis. After that, we generate the component architecture as an XMI file and create an instance of our OSGi meta-model using this file. The second sub-process titled *Dynamic Software Architecture Evolution*, its objective making the recovered architecture interactive which allows developers to update or removing existing components or adding new ones during runtime. It starts with a graphical visualization of software architecture which helps developers to understand and comprehend the system to start the evolving and maintaining task. We can work with an existing software architecture or with the recovered one from the previous process. Next, developers can begin the evolution task and updating the system via the architecture, they can add, update, remove components, or update its properties at runtime. In the end, we get new software architecture and an evolved system. We implemented our proposed process on a set of tools integrated into Eclipse. These tools offer: i) recovering software architecture at runtime with all its components, so that assist the comprehension of the system. ii) a graphical editor for visualization and analyzing the recovered software architecture at runtime, so that provides a clear and abstract representation helps to understand the system with less time and effort. iii) evolving the system graphically using software architecture, which makes this task easier and faster. iv) generating automatically the new software archi-

tecture after the maintenance and evolution task.
We plane in the near future to:

- Experience our app with platforms other than Eclipse to demonstrate its effectiveness.
- Evaluate the aspect of our application in recovering the software architecture using different techniques.
- Evaluate the aspect of generating the new system after the task of evolving its architecture.

Bibliography

[Akm+17] Feidu Akmel et al. "A Comparative ANALYSIS ON SOFTWARE ARCHITECTURE STYLES". In: *International Journal in Foundations of Computer Science & Technology* 7.5/6 (2017), pp. 11–22. DOI: 10.5121/ijfcst.2017.7602.

[Ale] Alexander.D.Brown. *OSGi Demystified: 5.1 – Declarative Services: A Tutorial*. URL: https://developer.ibm.com/cics/2019/01/29/osgi-declarative-service-tutorial/.

[Alla] Architecture – OSGi™ Alliance. *The Dynamic Module System for Java, What is OSGi?* URL: https://www.osgi.org/developer/what-is-osgi/.

[Allb] OSGi™ Alliance. *OSGi Core Release 7*. URL: https://docs.osgi.org/specification/osgi.core/7.0.0/framework.lifecycle.html.

[Allc] OSGi™ Alliance. *The Dynamic Module System for Java, Architecture*. URL: https://www.osgi.org/developer/architecture/.

[Alld] OSGi™ Alliance. *The Dynamic Module System for Java, Benefits of Using OSGi*. URL: https://www.osgi.org/developer/benefits-of-using-osgi/.

[AME12] Aitor Agirre, Marga Marcos, and Elisabet Estévez. "Distributed applications management platform based on Service Component Architecture". In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE. 2012, pp. 1–4.

[ASS14] Seza Adjoyan, Abdelhak-Djamel Seriai, and Anas Shatnawi. "Service identification based on quality metrics object-oriented legacy system migration towards soa". In: *SEKE: Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School. 2014, pp. 1–6.

[BA+15] Syed Mohtashim Abbas Bokhari, Farooque Azam, et al. "Limitations of Service Oriented Architecture and its Combination with Cloud Computing". In: *Bahria University Journal of Information & Communication Technologies(BUJICT)* 8.1 (2015).

[Bar12] Jeffrey M. Barnes. "NASA's Advanced Multimission Operations System: A Case Study in Software Architecture Evolution". In: *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '12. Bertinoro, Italy: Association for Computing Machinery, 2012, pp. 3–12. ISBN: 9781450313469. DOI: 10.1145/2304696.2304700. URL: https://doi.org/10.1145/2304696.2304700.

[Bar13] Jeffrey M. Barnes. "Software Architecture Evolution". In: 2013.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[BCL12] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. "Software architecture evolution through evolvability analysis". In: *Journal of Systems and Software* 85.11 (2012), pp. 2574–2592.

[BGS14] Jeffrey M Barnes, David Garlan, and Bradley Schmerl. "Evolution styles: foundations and models for software architecture evolution". In: *Software & Systems Modeling* 13.2 (2014), pp. 649–678.

[Blo] Vogella Blog. *Getting Started with OSGi Declarative Services*. URL: http://blog.vogella.com/2016/06/21/getting-started-with-osgi-declarative-services/.

[BR00] Keith H. Bennett and Václav T. Rajlich. "Software Maintenance and Evolution: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: Association for Computing Machinery, 2000, pp. 73–87. ISBN: 1581132530. DOI: 10.1145/336512.336534. URL: https://doi.org/10.1145/336512.336534.

[Bru+04] Eric Bruneton et al. "An open component model and its support in java". In: *International Symposium on Component-based Software Engineering*. Springer. 2004, pp. 7–22.

[Cen] IBM Knowledge Center. *Example: OSGi bundle manifest file*. URL: https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.osgi.nd.multiplatform.doc/ae/ra_bundle_mf.html.

[Cha+08] Sylvain Chardigny et al. "Extraction of component-based architecture from object-oriented systems". In: *Seventh working IEEE/IFIP conference on software architecture (WICSA 2008)*. IEEE. 2008, pp. 285–288.

[Con+09] R Conery et al. *Microsoft Application Architecture Guide*. 2nd ed. Microsoft Press, 2009.

[Crn+11] Ivica Crnkovic et al. "A Classification Framework for Software Component Models". In: *Software Engineering, IEEE Transactions on* 37 (Nov. 2011), pp. 593–615. DOI: 10.1109/TSE.2010.83.

[ESH10] Alae-Eddine El Hamdouni, Abdelhak-Djamel Seriai, and Marianne Huchard. "Component-based architecture recovery from object oriented systems via relational concept analysis". In: *CLA: Concept Lattices and their Applications*. 672. University of Sevilla. 2010, pp. 259–270.

[FA04] Paolo Falcarin and Gustavo Alonso. "Software architecture evolution through dynamic aop". In: *European Workshop on Software Architecture*. Springer. 2004, pp. 57–73.

[Fal+04] Katrina Falkner et al. "Unifying static and dynamic approaches to evolution through the compliant systems architecture". In: *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. IEEE. 2004, 9–pp.

[FOU] ECLIPSE FOUNDATION. *FAQ What is the plug-in manifest file (plugin.xml)?* URL: https://wiki.eclipse.org/FAQ_What_is_the_plug-in_manifest_file_(plugin.xml)%5C%3F.

[Gar00] David Garlan. "Software Architecture: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: Association for Computing Machinery, 2000, pp. 91–101. ISBN: 1581132530. DOI: 10.1145/336512.336537. URL: https://doi.org/10.1145/336512.336537.

[Geya] Carol Geyer. *Service Component Architecture (SCA) | OASIS Open CSA*. URL: http://www.oasis-opencsa.org/sca.

[Geyb] Carol Geyer. *Service Component Architecture Assembly Model Specification v1.1*. URL: https://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec-v1.1-csprd03.html.

[Gmb] 2020 vogella GmbH Lars Vogel (c) 2008. *OSGi Modularity - Tutorial*. URL: https://www.vogella.com/tutorials/OSGi/article.html#introduction-into-software-modularity-with-osgi.

[Gri] Duena Griego. *Software Evolution*. URL: https://www.slideserve.com/duena/software-evolution.

[Gro] Richard Gronback. *Eclipse Modeling Project | The Eclipse Foundation*. URL: https://www.eclipse.org/modeling/emf/.

[GS09] David Garlan and Bradley Schmerl. "Ævol: A tool for defining and planning architecture evolution". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 591–594.

[GSK14] Marvin Grieger, Stefan Sauer, and Markus Klenke. "Architectural restructuring by semi-automatic clustering to facilitate migration towards a service-oriented architecture". In: *2nd Workshop Model-Based and Model-Driven Software Modernization*. 2014, pp. 44–45.

[Haa] Johan den Haan. *What every architect should now know about the Service Component Architecture (SCA)*. URL: http://www.theenterprisearchitect.eublog/2009/03/11/what-every-architect-should-now-know-about-the-service-component-architecture-sca/.

[Hal+11] Richard S Hall et al. *OSGi in Action, Creating Modular Applications in Java*. Wiley India Pvt. Limited, 2011. ISBN: 9781933988917.

[HC01] G Heineman and W Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Jan. 2001.

[HMY04] Gang Huang, Hong Mei, and Fuqing Yang. "Runtime software architecture based on reflective middleware". In: *Science in China Series F: Information Sciences* 47.5 (2004), pp. 555–576. DOI: 10.1360/03yf0192.

[HQO16] Adel Hassan, Audrey Queudet, and Mourad Oussalah. "Evolution style: framework for dynamic evolution of real-time software architecture". In: *European Conference on Software Architecture*. Springer. 2016, pp. 166–174.

[Ker+19] M. L. Kerdoudi et al. "Recovering Software Architecture Product Lines". In: *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2019, pp. 226–235.

[KTS18] Mohamed Lamine Kerdoudi, Chouki Tibermacine, and Salah Sadou. "Spotlighting use case specific architectures". In: *European Conference on Software Architecture*. Springer. 2018, pp. 236–244.

[Lea] O'Reilly Online Learning. *Building Evolutionary Architectures*. URL: `https://www.oreilly.com/library/view/building-evolutionary-architectures/9781491986356/ch01.html`.

[LR02] Meir M Lehman and Juan F Ramil. "Software evolution and software evolution processes". In: *Annals of Software Engineering* 14.1-4 (2002), pp. 275–309.

[Lut+17] Thibaud Lutellier et al. "Measuring the impact of code dependencies on software architecture recovery techniques". In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 159–181.

[LZN04] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. "Applications of clustering techniques to software partitioning, recovery and restructuring". In: *Journal of Systems and Software* 73.2 (2004), pp. 227–244.

[MAV10] Jeff McAffer, Simon Archer, and Paul VanderLei. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 2010. ISBN: 9780321561510.

[MFP06] Nazim H Madhavji, Juan Fernandez-Ramil, and Dewayne Perry. *Software evolution and feedback: Theory and practice*. John Wiley & Sons, 2006.

[ML16] Hong Mei and Jian Lü. "Runtime recovery and manipulation of software architecture of component-based systems". In: *Internetware*. Springer, 2016, pp. 115–138.

[OLB16] Flavio Oquendo, Jair Leite, and Thais Batista. *Software Architecture in Action*. Springer, 2016.

[OST05] Mourad Oussalah, Nassima Sadou, and Dalila Tamzalit. "A generic model for managing software architecture evolution". In: *Proceedings of the 9th WSEAS International Conference on Systems*. World Scientific, Engineering Academy, and Society (WSEAS). 2005, pp. 1–6.

[Pér+05] Jennifer Pérez et al. "Dynamic evolution in aspect-oriented architectural models". In: *European Workshop on Software Architecture*. Springer. 2005, pp. 59–76.

[PPR03] C. Pérez, T. Priol, and A. Ribes. "A Parallel Corba Component Model for Numerical Code Coupling". In: *The International Journal of High Performance Computing Applications* 17 (2003), pp. 417–429. DOI: 10.1177/10943420030174006.

[Rah] Dewi Rahmawti. *Evolution processes*. URL: http://share.its.ac.id/blog/index.php?entryid=968.

[Rou] Margaret Rouse. *What is Enterprise JavaBeans (EJB)? - Definition from WhatIs.com*. URL: https://www.theserverside.com/definition/Enterprise-JavaBeans-EJB.

[Ryc17] Marek Rychly. "Dynamically reconfigurable architectures: An evaluation of approaches for preventing architectural violations". In: Jan. 2017, pp. 539–556. DOI: 10.4018/978-1-5225-3923-0.ch022.

[Szy02] Clemens Szyperski. "Component Software: Beyond Object-Oriented Programming". In: (2002).

[Tib] Chouki Tibermacine. *Programmation par composants avec osgi*. 2013-2014. URL: http://www.lirmm.fr/~tibermacin/ens/gmin30f/cours/cours2.pdf.

[Tuta] Tutorialspoint. *EJB - Overview - Tutorialspoint*. URL: https://www.tutorialspoint.com/ejb/ejb_overview.htm.

[Tutb] Tutorialspoint. *Software Engineering Overview - Tutorialspoint*. URL: https://www.tutorialspoint.com/software_engineering/software_engineering_overview.htm.

[Wik] Wiki.eclipse.org. *Graphical Modeling Framework - Eclipsepedia*. URL: https://wiki.eclipse.org/Graphical_Modeling_Framework.

[WOR] EJB WORLD. *Day 02. Understanding EJB Types and Interfaces – EJB In 21 Days ?* URL: https://ejbvn.wordpress.com/category/week-1-enterprise-java-architecture/day-02-understanding-ejb-types-and-interfaces/.

[XZ10] Hongzhen Xu and Guosun Zeng. "Specification and verification of dynamic evolution of software architectures". In: *Journal of Systems Architecture - Embedded Systems Design* 56 (Oct. 2010), pp. 523–533. DOI: 10.1016/j.sysarc.2010.08.005.