DESIGN PATTERNS USING COMPONENT-BASED SOFTWARE DEVELOPMENT

A dissertation submitted to The University of Manchester for the degree of Master of Science by Research/Master of Enterprise

In the Faculty of Computer Science

2009

Arsalan Sadri

School of Computer Science

Contents

1. TABLE OF FIGURES

2. FORMATTING AND KEYWORDS

FORMATTING

With respect to the editing, the following formats are used in the dissertation:

- Page margin; top: 2.8 bottom: 2.25 left: 4 right: 2.4
- The body; font: Verdana, size: 10 spacing: double
- Headings; font: Arial, size: 16(bold), 14, 12, 11 type: capital letters
- Header and footer; font: Verdana, size: 8, type: bold, italic spacing: single
- Quotations; font: Verdana, size: 10, type: italic, spacing: single
- "Note:" comments: font: Verdana, size: 10, type: italic, spacing: single
- Caption for figures: font: Verdana, size: 9, type: bold
- Referencing: type: numerical, ISO 690 in Microsoft Word 2007, automatic insertion

KEYWORDS

- Component-based software development, Component model
- Design patterns, Composition operator, Exogenous connector
- Atomic/Composite Connector, Atomic/Composite Component
- Reusability, Repository
- Object, Class, Inheritance, Composition, Interface, generic

3. ABSTRACT

Design patterns are used in object-oriented programming when a problem and its solution fit a well-known pattern, i.e. the pattern provides the solution to the problem. It would seem that design patterns should be just as useful for component-based development. Indeed, in component-based development, design patterns are potential composition operators. This project will investigate design patterns that will work as composition operators for components, and define and implement them.

The following is the set of achievements in this dissertation:

- Understanding, learning, and applying fundamental principles of object-oriented programming
- Learning and applying some basics and hints of Java programming language which I was not familiar with
- Getting familiar with Eclipse IDE tool
- Understanding, learning, and applying basic principles of component-based software development, particularly the concept of reusability since it is a key topic in this project, as far as it applies to my area of research
- Understanding, learning, and applying the notion of design patterns in object-oriented programming
- Investigating the problem of using design patterns in current models(object-oriented paradigm)
- Figuring out how this problem can potentially be solved by redefining design patterns (but keeping the original meaning, structure, and application the same) as composition operators in component-based approach
- Implementation of above-mentioned solution

4. DECLARATION

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

5. COPYRIGHT STATEMENT

(1) The author of this dissertation (including any appendices and/or schedules to this dissertation) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

(2) Copies of this dissertation, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

(3) The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

(4) Further information on the conditions under which disclosure, publication and exploitation of this dissertation, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of Computer Science.

6. ACKNOWLEDGEMENT

I would like to take this opportunity to acknowledge my supervisor Dr. Kung-kiu Lau who inspired me to carry out this research project.

7. INTRODUCTION

First, I go through the notion of component-based software development. I also briefly cover issues facing components reusability and requirements for software reuse. Next, I elaborate on imperfections existing in current component models and then solutions are explained on how to overcome those problems. I try to have a quick review of some aspects of this notion which widen my horizon so that I can establish a connection between component-based development and design patterns.

I explain The Component Model (1) and its fundamental principle and that how it can be employed to implement design patterns for reusability purposes. After that, I argue what the drawback of using patterns in the context of object-oriented programming is and how it force us to come up with another view on design patterns to overcome this drawback. To be more exact, the current drawback of using design patterns in the context of object-oriented programming is that in this way design patterns are not implemented and hard-coded such that they can be reused in the future, i.e. patterns are just explained in a formal way; however, they are neither coded nor stored and they have to be implemented each time. Thus, they cannot be reused in reality. I argue that to be reusable and to achieve code reuse they need to be brought into the context of component-based development so that they can be stored as a component (composition operator or composite connector in Component Model) in a repository and be reused for future purposes. To illustrate this in practice, I have implemented and composed two design patterns, Observer and Chain of Responsibility, using fundamental concepts of Component Model. This approach towards design patterns, however, cannot be applied to all patterns.

In fact, design patterns are standardized and are used in the context of object-oriented programming; however, they do not include a concrete implementation

but just abstract descriptions of what they are, how they can be used, how they solve problems. After that, I walk you through the fact that how patterns can be used or must be used (to achieve code reuse) in the component-based software development paradigm, particularly in The Component Model.

Note: This component model will be called 'The Component Model' from now on in this dissertation.

8. COMPONENT-BASED SOFTWARE DEVELOPMENT

8.1. MOTIVATIONS

Component-based development is a relatively a new area in the field of software engineering, promising of a design strategy facilitating the task of software development. In fact, the elements that Component-based development provides support this design strategy. Component-based development (2) can be described as the integration of previously existing software components. In simple words, Component-based development aims to investigate, design, and implement large-scale applications using code reuse, that is, by employing pre-built components. It relieves system analysts from the tedious task of redesigning and recoding systems by componentising existing packages and processes. This has a significant effect on the development and modification of systems in the future especially in large-scale enterprise system. Components are designed either for business processes or standard scientific application. For instance, using Component-based development approach An ERP, Enterprise Resource Planning, offer variety of packages such as manufacturing, supply chain management, financial, human resource management, and so on which suits different processes of an enterprise. Typically an enterprise chooses from these packages in accordance with its business needs. In this way ERP builds an efficient platform upon which homogenous interacting software components will be assembled.

"If components are developed independently, it is highly unlikely that they will be able to cooperate usefully." (2)

Now imagine what would happen in the absence of components. To satisfy new needs and business processes, efforts would have to be made from very preliminary stages and large number of programmers would be needed to handle all tasks. In contrast, using Component-based development, new components

which fulfil new tasks may be built from composition of previously-designed, previously-implemented and previously-tested components. This results in robust products. It is the beauty of component-based approach that system developers can benefit from yields of others. This approach is time-saving, cost-effective, and maintenance-reducing. It also largely reduces the number of human resources needed for designing and coding components, resulting in more productivity.

8.2. CONCEPT OF COMPONENT

The concept of component is loosely defined. In fact, as far as I am concerned, there is no universally-accepted definition. Each definition tries to give an exact picture of what a component is and how it functions. In figure 1, 2, and 3, a component is depicted from different perspectives. In figure 1, you can see an abstract picture of a component with an interface that enables it to interact with other components to which it is compatible. In the following chapters, I elaborate more on what is meant by an interface.

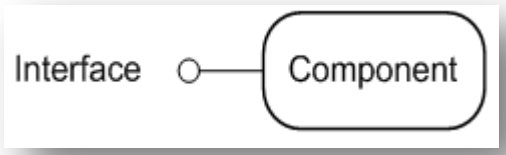

Figure 1: A component with an interface

As I mentioned earlier, there is no universally-agreed definition of what a component is. All of them, however, have something in common: components as reusable templates. At this point, I would like to briefly refer to some of those definitions:

"A component is a unit of software of precise purpose sold to the application developer community." (3)

"A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort." (4)

- A component (5) (6), which normally models real word objects, adheres to a set of interfaces as a contract, performing a distinctive set of activities known as the functionality of this component and doing so enforce components to behave systematically. From this behaviour which is determined by component's contract, developers can figure out whether a particular component can communicate with others both at design and deployment phase.
- Components are reusable software elements than can be matched together like a puzzle (figure 2). In fact, they are building blocks of a larger system.

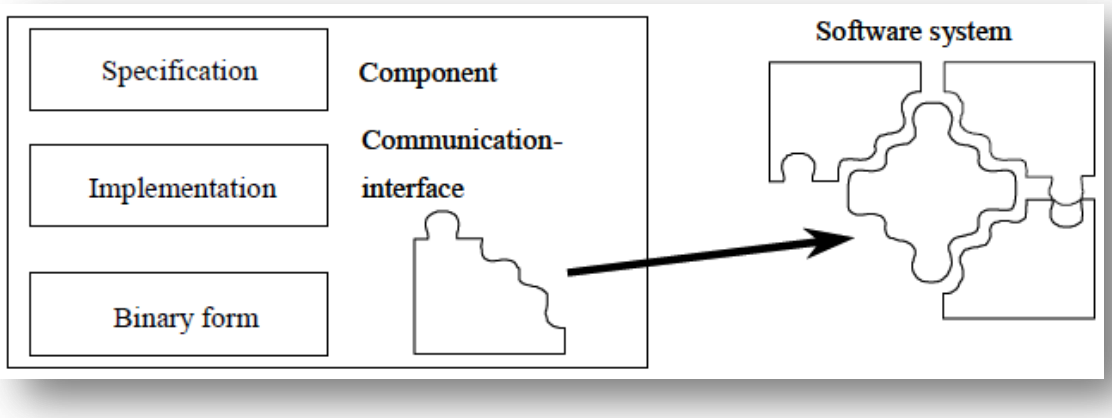

Figure 2: A software component like a puzzle

8.2.1. PROPERTIES OF A COMPONENT

The most important property of a component might be its reusability feature. This becomes even more tangible when users of components are not the same as their developers. (7) In this case, the developers need to design them such that they can be used and deployed by customers.

To perform a task a component needs to be bound to a resource described by component models. Binding of components implies component composition. Three forms of compositions have been identified: component-component, component-framework, and framework-framework (6) . Each of them addresses communications between components, components and frameworks, and frame works respectively. To put it simply, composition states how components work together. As it seems, one pre-requisite to achieve this goal is that components need to be aware of provided and required services of others.

A component is tightly bounded to the component model in which it is designed, implemented, and deployed. A component's properties, therefore, stem from those of its component model. As an example, design rules, which minimises any type mismatch among components, are specified by component models and components have to agree with these rules. I talk about component properties in more details in the following chapters.

8.3. SOFTWARE REUSE

One of the indispensible features of components is their reusability. Components are developed systematically in a component models which enforces a set of behaviours; thus, if a component cannot be reused it is no longer considered to be a component.

Note: Reuse does not necessarily mean that these reusable elements would be applied to a new application without any modification. Indeed, after being selected from existing components, selected components need to be tailored so as to fit into target applications. This can imply modification.

8.3.1. SOFTWARE REUSE REQUIREMENTS

8.3.1.1. DOCUMENTATION

The documentation should faithfully represent the whole functionality of interfaces.

8.3.1.2. INTERFACES

The interface of a component is a visible contact point to outside world, making users unaware of inside of components and their implementation. The interface of components also describes components' type, which in turn reflects its role and functionality in a given system. Components can have more than one interface; it means that they conform to more than one contract, eventually resulting in a set of approved behaviours. To be more specific, when a component implements multiple interfaces, it automatically abides by multiple operations. More detailed, each interface acts as an interactive language for components to communicate with outside world, that is, other components. This leads to one of the key principles of Component-based development and object-oriented programming: there is no way to know about a component or an object except from its interface. Such ability is known as encapsulation with respect to programming issues. Naturally, each of these types of a component is then utilized in an appropriate application it applies to. Component interface needs to be documented well so as to make their usage more convenient. Recall that not all languages support multiple interfaces. Some programming languages (Java) allow and some others disallow usage of more than one interface.

8.3.1.3. CONCEPTUALIZATION

Another requirement is conceptualization of components as well as their interactions with their environments. That is to say, developers need to equipped components with a level of abstraction such that it makes components understandable without recourse to knowing detailed codes.

8.3.1.4. CODE HIDING

With respect to proprietary issues, one of the requirements of component-based software development is code hiding. It means that developers need to conceal

their codes and implementation from being visible to public so that they cannot be abused.

8.3.2. SOFTWARE REUSE BARRIERS

8.3.2.1. FUNCTIONAL BARRIERS

One of the barriers that potentially erects when designing reusable elements is functional barriers. Let me first make it clear by making an example. Imagine installing the GPS system (Global Positioning System) on vehicles. There are two points to be considered; one, the type of vehicles that the GPS system is to be installed on and second, the number of services a particular GPS application offers. Cares need to be taken to choose a sound application so as to avoid imposing incompatible or excessive services on the system, i.e. a particular GPS application which would be installed on airplanes would have too many complicated services for a simple vehicle. With respect to functional barriers, all I am trying to convey is that not all component-based applications can be integrated into a new environment as separate building units. As a result, some services may not function properly.

8.3.2.2. PLATFORM-ARISING BARRIERS

Among other difficulties, I can refer to platform barriers. Nearly all components are designed and then implemented in a particular programming language, a certain platform, and using a set of data structures. Therefore, there is likelihood that they cannot be deployed into another environment. For example, a component written in .NET may not be integrated in a component models written in Java. Similarly, different platforms stores information differently such that a component working in a certain platform may not perform the same tasks in another platform.

"They cannot be used because the chosen parts do not fit together." (2)

If I can just sum up the main points, when developing a new system using component-based development, an exhaustive consideration need to be taken to make sure components which will be applicable and efficient for the new system would be selected. This is referred to as domain engineering. (2)

8.4. COMPONENT MODELS

8.4.1. AN OVERALL LOOK

Component models can be viewed as a collection of components, their types (interfaces), and their relations to one another. They determine rules that components need to comply with. In this manner, components are like extension to the whole system. A component model functions like a standard agreement for its components, providing basic infrastructure and services for components' design, implementation, composition, and deployment. This improves predictability. There is no universal agreement on what elements are needed to make up a component model. Figure 3 shows a general view of a component model including various parts of the system. Let's now look at some of the terms mentioned in this figure.

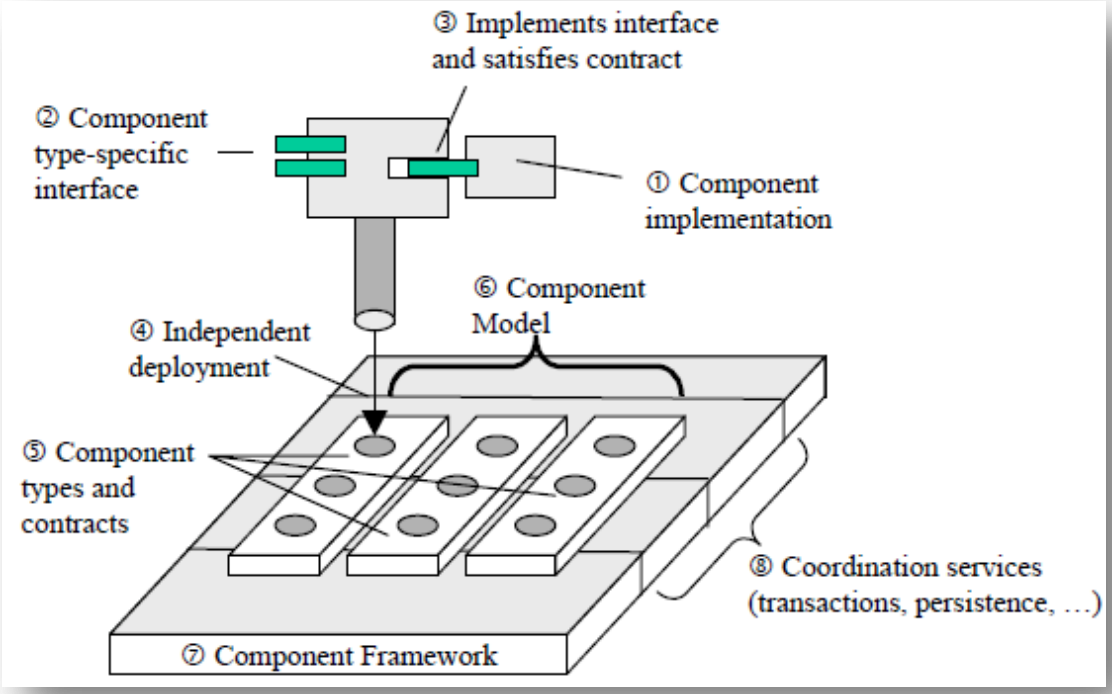

Figure 3: A component model

Component types (black holes in figure 3) make up a component model. As mentioned earlier, component's type is determined by its interface. A component framework offer different types of runtime services for deployment. The design environment should be consistent with deployment environment so that pre-built components can be deployed and be run. This lies in component's framework platform. A component framework functions similarly to an operating system, but in smaller scale. (6) They are nearly enabled to do whatever an operating system may do such as starting and terminating components, establishing connection between them, and managing shared resources. For example, the Enterprise JavaBeans supports The EJB component model by providing servers and containers as a framework, where containers deal with components' lifecycle and servers address different required services.

But, how components can connect together? This introduces the notion of connectors which is of interest of this dissertation. Now let's have a quick look to

definition of connectors in different component models. As a standard, the connection should be in proper way. In terms of semantic, components can be considered to be a computational unit with some in and out ports for required and provided services respectively. There are various types of connectors in different component models. For instance, in UML2 and CCM these ports that designed for required services are distinct from those of provided service, thus playing a separate role. Another well-known component model is JavaBeans that employs a container and an Adaptor class, to provide a mechanism to establish a connection between communicating components (beans). In ADL, Architecture Description Language, connectors are a part of components and handle transformation of data between components. This transformation is achieved in Coordination Languages through a set of compositional channels. In general, two tasks need to be performed in any component models, first computation and second communication.

As it can be realized, in component models mentioned above, there are no distinct entities to perform these 2 tasks. In fact, all of them address both communication and computation, yet they do not provide separate entities to accomplish this goal. But, why not having separate entities might be important at all? In other words, what is the problem of not having such separate entities?

Let me examine this by analysing how components in practice communicate with one another and they send messages. There are two ways for message sending, direct and indirect.

The former (figure 4) uses direct method call to invoke other methods, that is, sending message would be directly from the sender component to the receiver. This kind of connection has two drawbacks. First, it mixes computation with communication as connectors are a part of components. Second, it may increase

coupling between pair of corresponding components. High coupling is not desirable whatsoever since it heighten dependency between components.

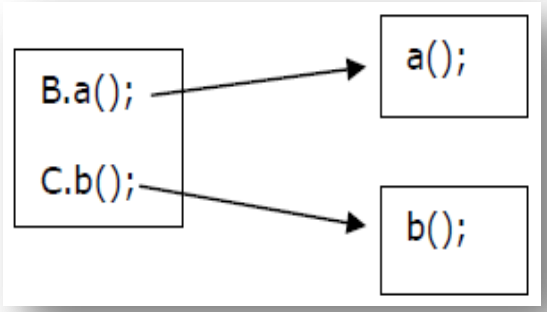

Figure 4: Direct Message Passing

The latter (figure 5), however, do not increase coupling as much as the former does because there would be some distinct entities to perform communication between components; therefore, resulting in separating communication from computation. In this way messages would not transmit directly from the sender to the receiver, rather they would be passed through these separate connectors. Moreover, direct message passing suffers from the problem of competing calls to the same component, which needs to be addressed by synchronizing components.

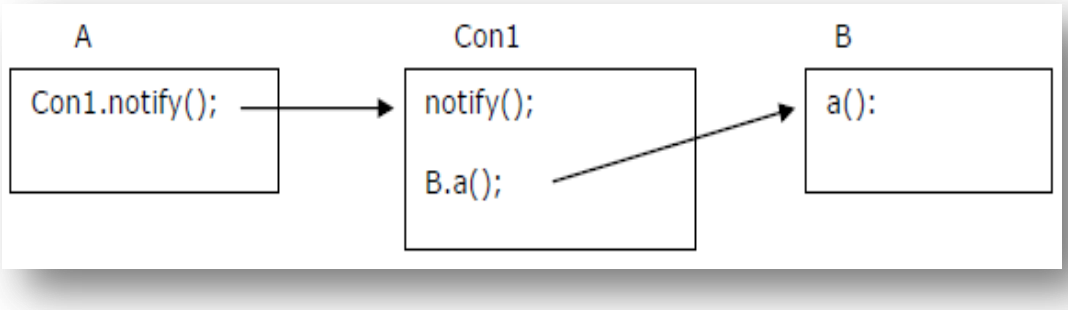

Figure 5: Indirect message passing

8.4.2. AN IDEALIZED COMPONENT MODEL

So far, we have looked at some of the basic issues in connection with components and component-based approach and I also mentioned a couple of concepts with respect to component models and component composition.

Let me turn now to some of the characteristics of an idealized component model. An idealized component model should address the following issues:

- Design, implementation, and deployment phases in consistent way
- Life-time of components as they are independent entities
- Providing required services for components
- Components composition
- Starting components, allocating memory to them, handling message passing, and terminating them

An idealized component model is based on assembly of components as truly reusable software elements. These reusability needs to be applied in practice when using and deploying components. It means that there must be a repository for components so that having been coded, they can be deposited into this repository and then be retrieved at the time of designing applications. In this way, components should be composable such that they can be composed into larger composite components, which in turn can be composed further. (7) To reach this set of ideals, an idealized component model may have two stages, one design phase, and second deployment phase, which fit to design and deployment of phases respectively.

8.4.3. THE COMPONENT MODEL

Note: As I previously mentioned in the introduction, The Component Model term refers to the component model developed in the University of Manchester under supervision of K.-K Lau. (1)

As it should be clear by now, we witness one common problems associated with all component models have been covered in previous sections. The problem is that computation is mixed with communication, i.e. there are no distinct entities that carry out these tasks separately, and rather messages are passed through entities which are also responsible for performing computation. In simple words, when a computation unit needs to call another method in another component, it does it directly. In addition to this, connectors do not have separate coding, but they are mixed with computation units. Now to remedy this, I would like to introduce a new component model (1) as a solution which will form the infrastructure for the final result of this dissertation, redefining patterns as composition operators. To be more exact, in the following sections, I demonstrate how design patterns are defined as composition operators based on this component model as underlying idea.

In this component model, Exogenous Connectors for Component Models, the distinguishing feature is encapsulation. It encapsulates computation in computation unites and communications in exogenous connectors. In other words, computation is performed and only performed in the computation units and communication is handled and only handled in exogenous connectors. This implies that connectors need to be independent entities and therefore, have segregate implementation. Since connectors are separate elements, they can therefore be stored in a repository for reusability purposes. The figure 6 shows how exogenous connectors initiate, manage, and terminate control flow. As you can see, components do not have any code to call other operations belonging to other components. Operations are merely invoked by exogenous connectors. As a result, a component, which is just a computation unit, is executed if and only if it is invoked by an exogenous connector.

In this component models, any desired application is built as a hierarchy of different levels of components and connectors in design phase, then in

deployment phase they are deployed, and finally they are assigned value at run time. This model, however, has a disadvantage of imposing preponderance of such levels. (1)

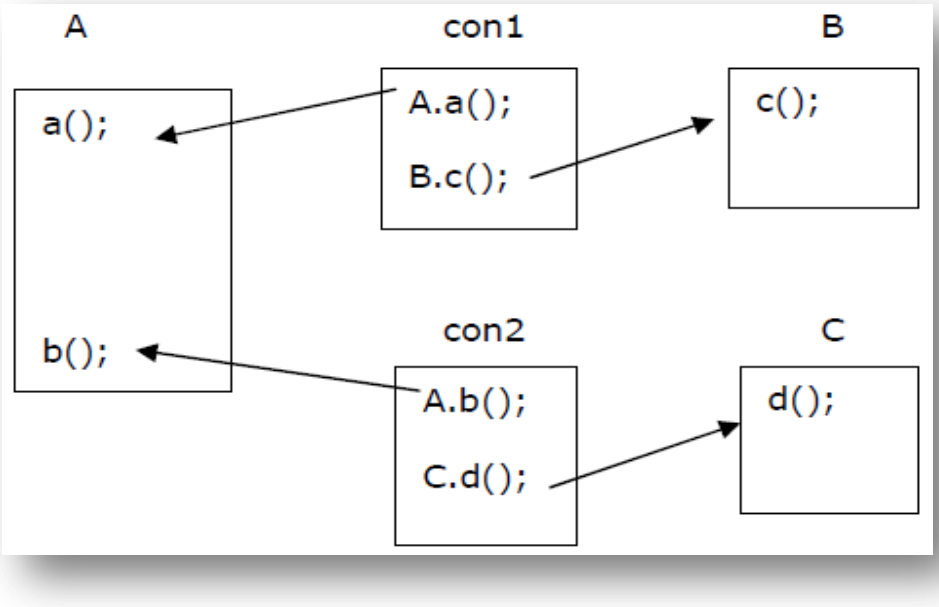

Figure 6: Message passing in The Component Model

Some supplementary examination is followed up at the end when I open up the main discussion of redefining patterns as composition operators.

9. DESIGN PATTERNS IN OBJECT-ORIENTED PROGRAMMING

9.1. OVERVIEW

Let's begin with a very brief explanation of two programming styles. Object-oriented paradigm is usually compared and contrasted with structural programming. The major philosophical difference is that in the structural programming the centre of focus is on functions (methods, behaviours, actions, operations, or procedures) whereas object-oriented paradigm in essence focuses on objects. This makes it possible for the latter to benefit from having everything in one entity, that is, the object. In fact, objects encapsulate both information, represented as fields, and methods, representing the set of operations that can be applied on these fields. An object of type Person for instance, may represent information such as name, address, age, eye colour, and nationality, and also some of behaviours of ordinary persons like eating, drinking, sleeping, chatting, and so on. Finally, the system would be an environment of communicating objects of type Persons. (Figure 7)

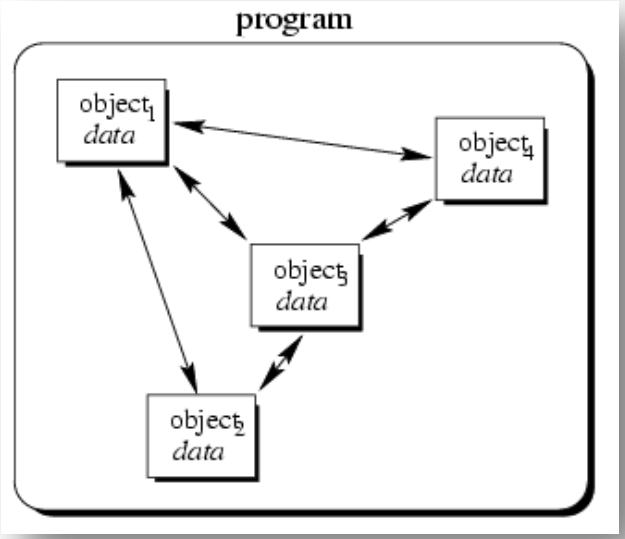

Figure 7: Objects interaction

9.2. ALL ABOUT PATTERNS

9.2.1. WHY DESIGN PATTERNS AT ALL?

I would like now to discuss why object-oriented programming might be a tedious task. Having identified system requirements as well as the system workflow (this process is subject to methodology used in kind of the selected software engineering approach), the key factor in object-oriented design is to break the system down into a well-established class hierarchy such that it can faithfully represent the behaviours of the system and accommodates data information. In addition to this, such class hierarchy needs to consider the potential developments of the system in the future. In simple words, since classes are related together, through inheritance and composition, a change in one class may require a change in others. As it can be realised, it would seem that such a process needs careful consideration as well as exhaustive efforts to make sure any possible modification in the future is now minimised.

Now the point is in many occasions developers face with the same problems which happen again and again in various applications. In this case, they come up with a solution, design it, code it, apply it, and then observe consequences. If needed, they modify their solutions for best performance. This process, facing the same problems and discovering solutions, appear to be like a loop that needs to be repeated by system designers. The point is the same algorithms tend to reoccur and this is an unwritten principle of software engineering. Now imagine each programmer wants to design, implement, and use their own solution, what would happen? They would end up with a set of inconsistent codes which cannot interact with one another since they were designed separately. Solutions would probably be intractable and hard to keep up to date.

Well, what is next? How to overcome such obstacle? This is where the design patterns come. They act like a solution template that can be reused over and over again provided that they are applicable, i.e. there should be a problem and its corresponding solution in a given context. A pattern describes a design idea, representing a reuse culture. This is welcomed in software development process and considered to be the most significant outcome of applying design patterns. Such culture minimises repetition of works. Such approach equipped developers with a higher level of perspective towards the system design, shifting their thoughts a way from details.

9.2.2. STANDARD DESCRIPTIONS

Design patterns are not defined formally and they are more like an abstract description of reoccurring problem solving issues, normally represented in terms of set of interacting objects or classes. At this point, I would like to draw your attention to some of definitions of design patterns in the literature.

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such

way that you can use this solution a million times over, without ever doing it the same way twice." (8)

"Design pattern is a formal way of documenting a solution to a design problem in a particular field of expertise." (9)

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design." (10)

"Patterns provide a means for capturing knowledge about problems and successful solutions in software development." (11)

As you may have noticed, all of these definitions have a concept to share and it is reusability concept. In the two following sections, titled with ingredients and purposes, I very briefly cover some of the elements constructing patterns and also what purposes patterns pursue. I then move to some of the guidelines on how to apply design patterns in practice.

9.2.3. INGREDIENTS

Ingredients of design pattern may be specified as the following:

- Name
  - Each pattern has a name. This should be descriptive enough to reminds us of patterns' intent
- Intent
  - What patterns do
- Problem
  - Each pattern matches to a reoccurring problem
- Solution
  - It clearly characterizes how a pattern solve the problem it fits to by identifying the whole factors including participants, their responsibilities, and their relationships

There is also another type of patterns, named architectural pattern, which is not of any interest to this dissertation. Architectural patterns, unlike design patterns, have larger scope and usually involve the whole systems (architectural level) rather than just a part of the system.

"Architectural patterns are software patterns that offer well-established solutions to architectural problems in software engineering." (12)

9.2.4. PURPOSE

Design patterns (8) can be organized in two ways. In terms of:

- Scope
  - Patterns deal with either objects or classes. This specifies the scope of patterns. The former (having object scope) is established dynamically through composition and the latter (having class scope) is achieved through class inheritance and therefore it has a compile-time level
- Purpose
  - This is indeed the intent of a pattern. It therefore states what patterns address. They can be grouped in three different categories namely, creational, structural, and behavioural patterns

9.2.4.1. CREATIONAL PATTERNS

They simply address object creation. Such patterns include Factory and Builder.

9.2.4.2. STRUCTURAL PATTERNS

They simply address object composition. Such patterns include Composite and Decorator.

9.2.4.3. BEHAVIUORAL PATTERNS

They simply address object interaction. Such patterns include Observer and Visitor.

9.2.5. HOW TO USE DESIGN PATTERNS

There are a couple issues that need to be considered when applying patterns. Normally, the story begins when a system analyst thinks he is facing with a reoccurring problem that might have a predefined solution. In this case, he needs to take into account some practises.

- What is the problem?
- Is there any potential pre-identified solution for this? If yes, can it be a design pattern?
- If it is a pattern, is there any other pattern that can be applied too?
- Is this pattern really more efficient than the simple solution in practice?
- Are there any inconsistencies with other elements in the system if this pattern is applied?

If a particular pattern was nominated to be applied in the system, developers [(8), (11)] then should:

(1) Go through the pattern to understand its usage as well as elements (intent, participants, and structure), and have a look to some sample codes to get familiar with its application in practice.

(2) Outline required classes as well as interfaces, and give them names appropriately

(3) Implement operations such that the set of responsibilities and object communication represented by this pattern is met accordingly

Since design patterns are intertwined with object-oriented programming, being familiar with professional practices in object-oriented design play a crucial role in exploiting design patterns.

Worth pointing that in (8), three practices are discussed in details as the core design principles in object-oriented programming and design patterns:

- design to interface
- favour composition over inheritance
- find what varies and encapsulate it

10. DESIGN PATTERNS USING COMPONENT-BASED APPROACH

10.1. SO, WAHT IS THE PROBLEM NOW!!?

By now, I have explained the necessary requirements, first the notion of component-based software development and second design patterns. You have seen on one hand, the essential elements composing component-based systems as well as their requirements, and on the other hand, you have got familiar with the fundamentals of design patterns. As it appears everything is going well and there is no barriers a head of the system design and development!!! Nonetheless, as I already signalled in previous chapters, if you more closely examine the design patterns, you will discover that although in abstract level they are reusable elements, which facilitate the process of software development, in practice they do not provide reusability since they are not hard-coded as a single entity whatsoever and therefore cannot be stored in a repository. To put it simply, they lack a formal definition that includes coding. In other words, there is not a universally-applied design and implementation structure. It means that if they were applicable in applications, each time they have to be coded from scratch for target applications and this approach would be subjective, as a result developers just share an abstract description of design patterns, nor a concrete implementation.

Note: There are some domain-specific patterns, which as their names suggest, they are just coded for a set of particular applications.

But why such formalization that results in concrete implementation is so significant at all? One of the benefits of such formalization is that it minimizes any vagueness, shifting the system developers' thinking away from complicated communication performed by patterns so that they can focus on the fixed behaviours of patterns in higher abstract level. It means that system designers

are no longer require to trace the internal communication algorithm of patterns, rather they simply can concentrate on the object interactions involved in a design patterns. Most importantly, I can refer to real reusability of patterns as the second benefit of such formalization. To be more explicit, in practice this may lead to using patterns as a single building entity in system construction since they are now independently designed and coded in a generic way and therefore can be used in any application. This implies that patterns need to be stored in a repository and then to be retrieved whenever needed.

There have been some efforts to formalize patterns and creating a compositional language which states this formalization. For instance, (13) suggests a method named Disco to model communications using classes, relations, and actions at abstract level. Figure 8 shows an example of Observer pattern formalization using this method. It is then followed by the combination of this with Mediator pattern to achieve more complex behaviour. This methodology, however, does not provide any coding and it merely suggests an abstract description using notation.

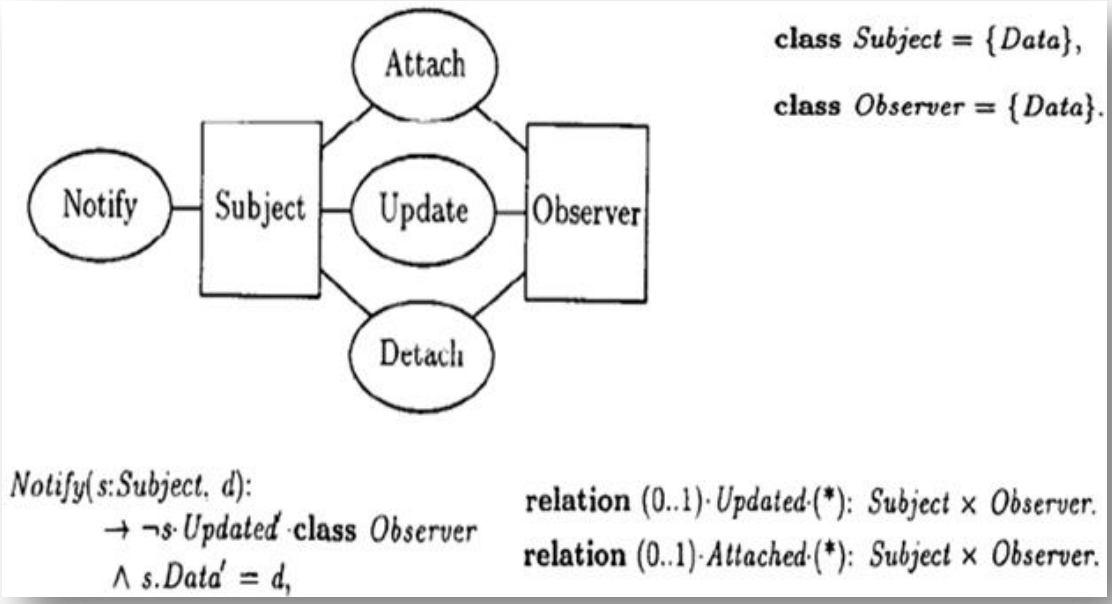

Figure 8: Formalization of Observer pattern in Disco

None of these efforts have managed to adopt an approach by which a pattern can be defined explicitly with concrete implementation so as to be deposited in repository.

10.2. SOLUTION: DESIGN PATTERNS AS CONCRETE REUSABLE ELEMENTS

10.2.1. PATTERN-LIKE COMPOSITION OPERATORES

We have looked at component-based software development principles and I briefly cover some of the issues relevant to the dissertation such as the concept of the component and its properties. I then mentioned some of the points regarding component models and I elaborated on the problems of current component models, which is not having separate entities, resulting in mixing computation with communication. This was followed by going through patterns and raising the problem associated with using patterns in object-oriented

programming, which is lacking formal definition, leading to not having concrete implementation.

By considering these 2 problems, I now want to offer a solution thereby a pattern can be defined and implemented as a separate entity and then deposited in repository. This solution is inspired by The Component Model I shortly talked about in previous chapters. In this model, a pattern can be defined, implemented, and stored in a repository as a composition operator, and then can be used indefinite number of times without requiring any modification of its implementation. This mechanism is generic and would work in any application provided that valid components are supplied, i.e. components that are applicable to the composition connector can attach to it. Additionally, bringing this solution into practice, we also remedy the problem of mixing computation and communication since in this model connectors are defined in a way they just manage the control flow and components just perform computation. Here, I do not go through preliminary concepts covered in previous section about The Component Model but I discuss the structure of The Component Model as it forms the basic structure of patterns that will be defined as composition operators.

Now let's see what is meant by a composition operator and why it can function like design patterns? To answer this, we need to analyse the ingredients of our solution which are connectors and components. Basically, the model is composed of connectors, computation units, and components. Computation units are in fact responsible for performing computation and in essence they are any files that can be executed and return a value, if any. For example, in Java programming language, they can be simple java classes with some methods performing distinct operations. There are 2 types of components, atomic and composite. (Figure 9) Before defining atomic and composite components, let's have a look at figure 9. As you can see, there is an entity named invocation connector. The invocation connector is responsible for invoking the required method inside computation

unit. This promises the fact that computation is separated from communication since no computation unit has any code to call methods in other computation units, that is to say, a method is executed if and only if an invocation connector invokes it and there is no other way.

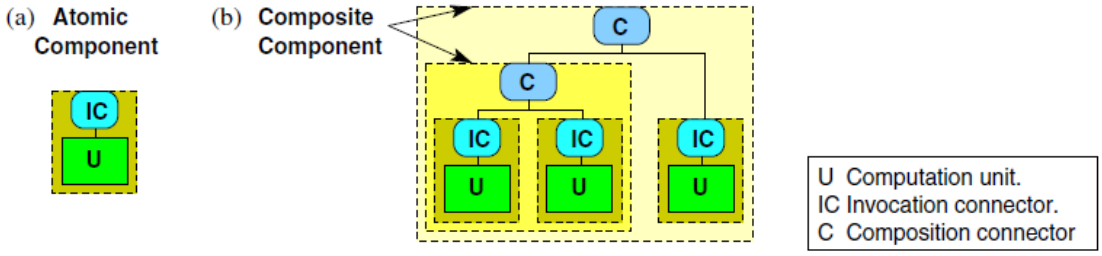

Figure 9: Atomic and composite components

An atomic component is composed of a computation unit as well as an invocation connector. A composite component is composed of a combination of atomic components and/or composite ones. This kind of construction shapes a hierarchy of component layers. Now there must be a mechanism to establish a connection among these layers. This is obtained by composition operators.

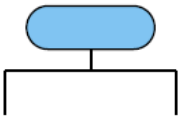

Figure 10: Composition operator

Note: The term composition operator is a general term and refers to connectors whether atomic or composite one. Also, a composition connector is another term for composition operator

They manage communication and handle control flow. In fact, they do not do anything, but passing data and invoking methods in proper sequence. Connectors like components can be atomic or composite. A composite connector is constructed using atomic ones. To manage control flow faithfully, the connectors need to represent ordinary control structures like sequencing, branching, and

looping. Such connectors are Pipe, Sequencer, and Selector. These connectors are placed on top of atomic and/or composite components to build the hierarchy levels. In figure 11, the computation units are labelled with alphabetical letters from A to G. As you can see, each computation unit has an invocation connector attached to it. Other connectors are then connected to these invocation connectors. When an application is modelled in this way, the whole system has one and only one top-level connector. For instance in figure 9, the highest connector, C, is the top-level connector which in turn has 2 sub-connectors, C and IC. The control flow always begins at top-level connector of any application. It would look like the 'main (args [])' method in Java language. In any Java-based application, the system would have one and only one main () method from which the system would be executed.

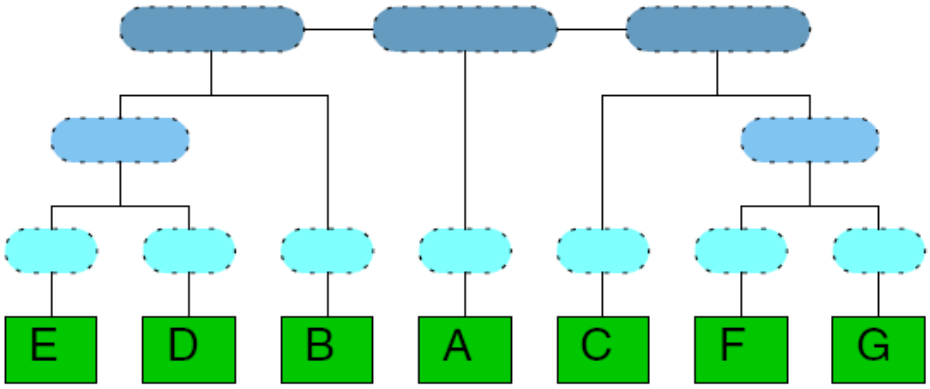

Figure 11: Hierarchy of components and connectors

Now I would like to discuss what is meant by a composite connector and why such a connector is of interest of this dissertation. I previously elaborated on atomic and composite components as well as atomic connectors. A composite connector is a combination of atomic and/or composite connectors. In other words, a set of interconnected atomic connectors forms a composite connector or a composite composition operator.

Note: From now on, for the sake of simplicity I will use the term 'composite connector'. It would be the same to replace it with' composite composition operator'

This composite connector can then be regarded as a separate entity exactly like an atomic connector. It therefore manages control flow, handles communication in hierarchy levels, and can be deposited in repository. Well, a question may have been raised in your mind is that if composite connectors function similarly to atomic ones (managing the flow of control), why do we have composite connectors then? What is the aim of having such connectors? The answer to this question is absolutely vital to this dissertation.

What we know from earlier sections is that a composite connector is a set of interconnected sub-connector. Thus, it performs more complicated control flow compared to atomic ones and it appears there is no boundary for composing composite connectors as far as complexity is concerned. That is, any number of atomic connectors can composed into a composite one. Here is the answer of our question. There must be intent for composing composite connectors. It means that since composite connectors represent a complex control structure, a composite connector is built whenever this control structure is desirable. Such control structures would be among commonly occurring control structures, i.e. design patterns. This is the basic idea and I go into details in next chapter.

10.3. APPLYING THE SOLUTION IN PRACTICE

10.3.1. OVEAL VIEW

In this section, I elaborate on how the abstract solution indicated in previous chapter can be applied in practice. In the previous chapters, I address the problem of using design patterns in the object oriented design and I also offer the solution to remedy this problem. The solution is that design patterns need to be applied in the concept of component-based development so that they can be

designed, coded, deposited into repository, and then deployed at run time. As I mentioned earlier, this promises code reuse since the user of such system would not make any changes to the source code of the components but they just compose components together in sensible way.

To achieve this goal, there must be a mechanism to redefine patterns in generic way so that they can work in any application provided that they were applicable in the given context. My work is inspired by The Component Model (1) that was described in abstract level in previous chapters. In such component model there are two basic entities and two basic phases. The former contains components and connectors and the latter compromises of design phase and deployment phase. At very first stage, computation units are designed and then stored somewhere. In the design phase, these computation units are then used for construction of atomic components. These atomic components are later on composed via composition operators which are connectors that manage control flow.

You saw earlier, a composite connector is obtained via composition of atomic and/or composite connectors and I briefly mentioned that there must be intent for composing connectors together. A composite connector reflects a composite control flow structure since they encapsulate control. This composite connector can be designed such that it represents a design pattern. Behavioural pattern in object oriented paradigm address interaction between objects and that how they communicate with each other. To exemplify, Observer pattern defines a one to many dependency between a (a set of) publisher(s) and subscribers in a manner that whenever the publisher is updated, the subscribers are notified and react accordingly. This fixed behaviour can potentially be represented by a composite connector that receive a request passes it through the publisher and receive the result back and pipe it through subscribers to notify them. In such a design, the composite connector that serves as a pattern is not aware of the type of the request being received and the type of the result being piped, it merely enables

the control flow and passes the result in a predetermined order. In the following chapters, I explain in details that how such a mechanism can be exploited in practice to enable a composite connector to play the role of behavioural patterns.

10.3.2. ATOMIC COMPONENT

Our component model compromises of components and connectors. As mentioned before, there are two types of components, namely atomic and composite. Composite components are composed of a set of atomic and/or composite components. As its name indicates, an atomic component is not a composite element and therefore, it does not rely on other components in terms of internal structure, i.e. it does not contain any other components. Atomic components are constructed when an invocation connector is bound to a computation unit. The figure 12 shows The UML diagram for AtomicComponent.java class.

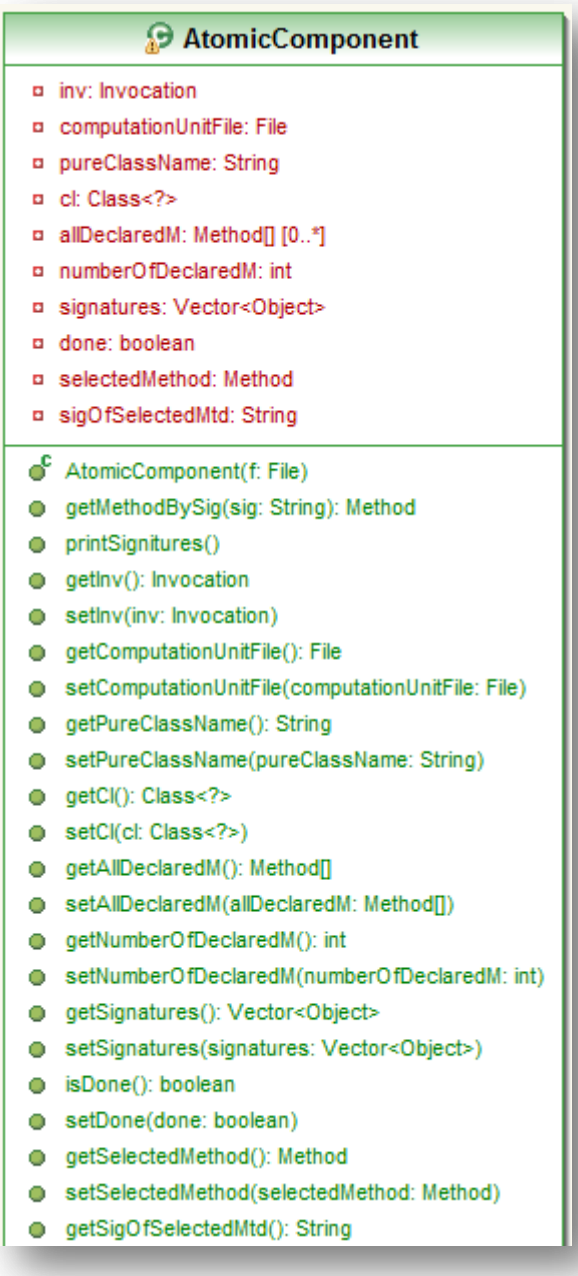

Figure 12: AtomicComponent.java

As it is clear, each atomic component has a reference to its invocation connector of type Invocation.java. Essentially, when an atomic component is instantiated using its constructor, this reference is assigned a value. The constructor receives an argument of type File that basically is the computation unit file selected by the

user. This file is normally selected from the repository through an interactive file chooser dialog box. The figure 13 shows an excerpt of this.

```
public AtomicComponent( File f )
{
	...
	Class< ? > cl = this.obtainClassObjOfCU( f );
	this.initAtomicComp( cl );
	...

}
private void initAtomicComp( Class< ? > c )
{
	...
	this.setCl( c );
	this.setAllDeclaredM( getCl().getDeclaredMethods() );
	this.setNumberOfDeclaredM( this.getAllDeclaredM().length );
	this.setInv( new Invocation( this.getCl().newInstance() ) );
	...
}
```

Figure 13:

As you can see at figure 12 and 13, atomic components have also references to other objects; most importantly, the reference to an object of type Class, which is used for invoking the desired operation (method) in the underlying computation unit. All in all, atomic component packages a computation unit and the invocation connector such that the invocation connector acts as a contact point to its corresponding computation unit, invoking one of its methods and returns the result back. If for some reason the invocation connector fails to execute the requested method, it returns the error has been occurred.

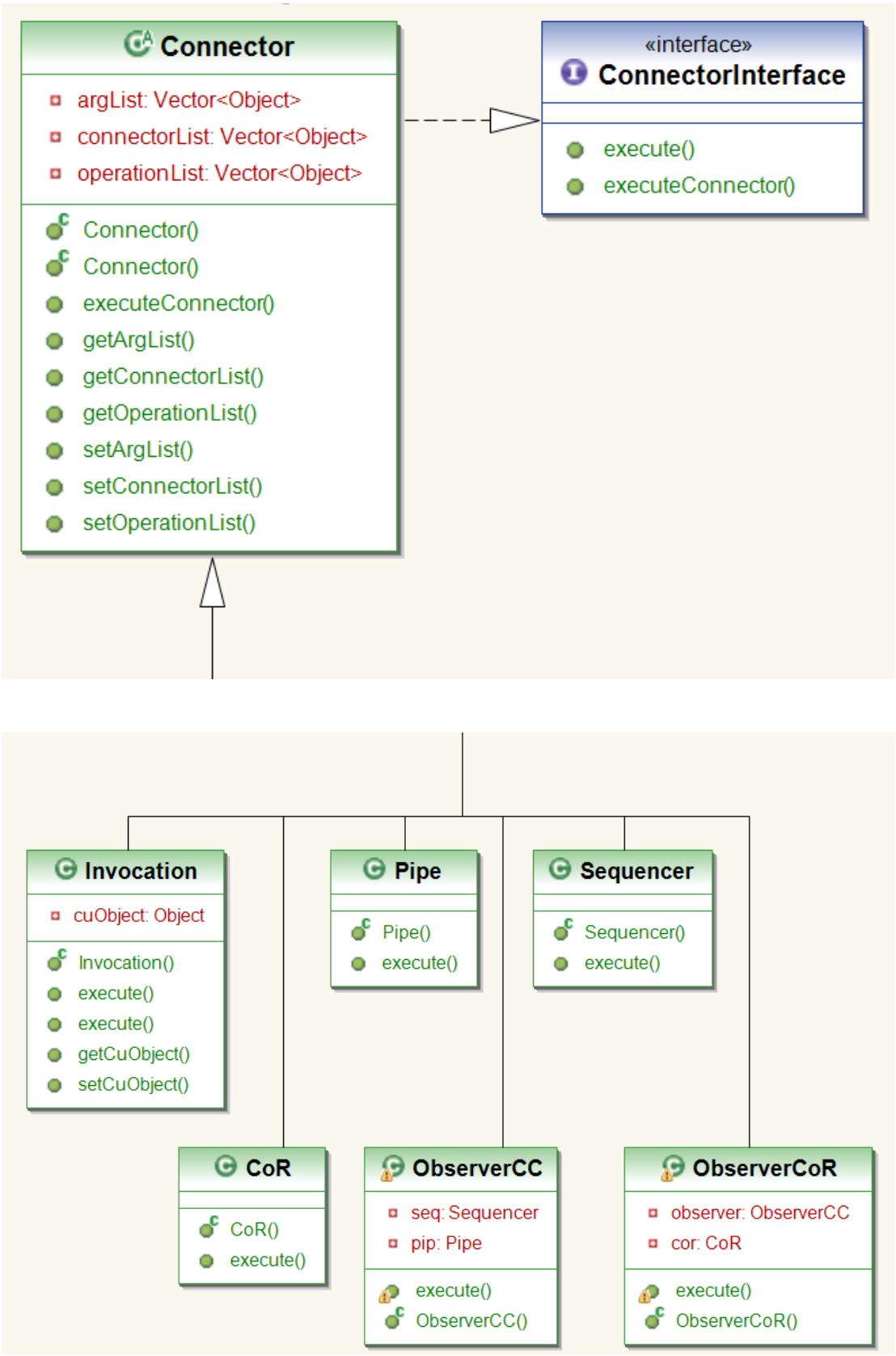

Figure 14: Class hierarchy for connectors

10.3.3. BASIC CONNECTORS

Genuinely, there are two types of connectors that construct our model. The first type is called basic/atomic connectors that can be of type invocation connector, pipe connector, and sequencer connector. Figure 14 shows an outline of the class hierarchy in the connector's package where all the connectors are defined and hard-coded. I define ConnectorInterface.java of type Interface, which enforces all types of connectors to perform two basic operations. It specifies execute() and exucuteConnector() methods that need to be performed by all connectors whether atomic or composite. Figure 15 shows a class named 'Connector.java'. This is basically an abstract class that defines those two basic operations as well as the data structure for all connectors and therefore all connectors need to extend this class as a super class.

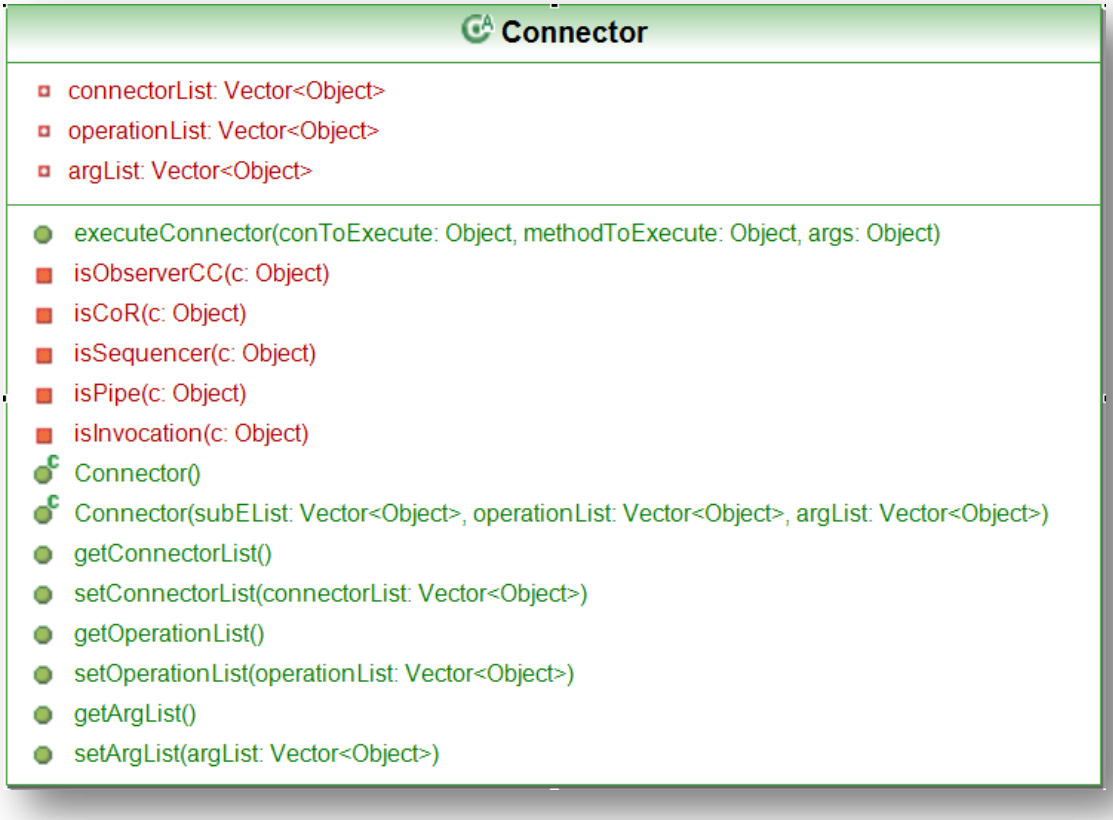

Figure 15: The Connetor.java class

A connector is run by calling its execute() method like conn1.execute();. Indeed, the execute method characterizes the functionality of each connector and therefore it needs to be overriden separately by different types of connectors.

However, the general outline of this method is almost the same for all connectors. Each composition connectors, has an interface. (Figure 16) The interface consists of three lists indicating the set of connectors attached to this composition operator, the set of operations to be run by these connectors, and parameters needed for running these operations. Those three lists are initialized through the constructor of each connector. (Figure 16) Thus, having been instantiated, a connectors knows what to do since it knows its sub-connectors and therefore, it just needs to call them by using their execute() methods. This means that we have an iteration of calling execute() method of each of sub-connectors. For instance, in figure 16, the first iteration would execution of c1 connector, which in turn invokes m1 operation using p1 parameter.

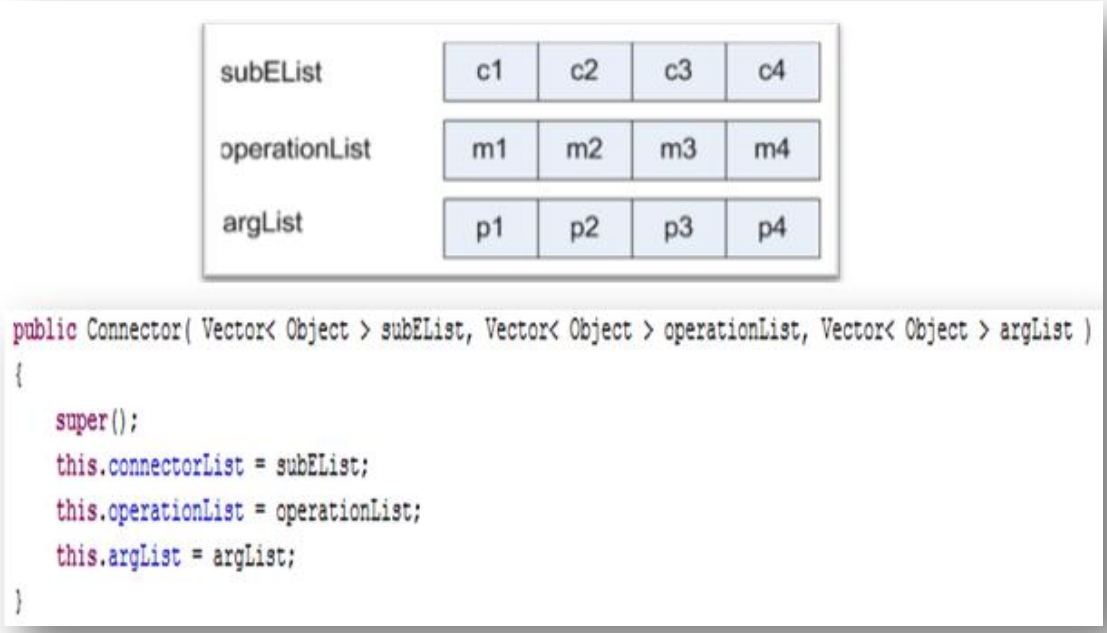

Figure 16: The interface of the connectors and its initialization

Figure 17 shows the control flows inside of a composite component with a connector as its interface. Having been called, the connector starts execution of its sub-connectors. (For simplicity we assume c1 to c4 are atomic components) As you can see in the figure 17, there is a predetermined order for execution of each sub-connector, from one to fourteen.

It is worth mentioning that the order is in fact determined when the connector is instantiated. Therefore, cares needs to be taken to ensure a proper instantiation.

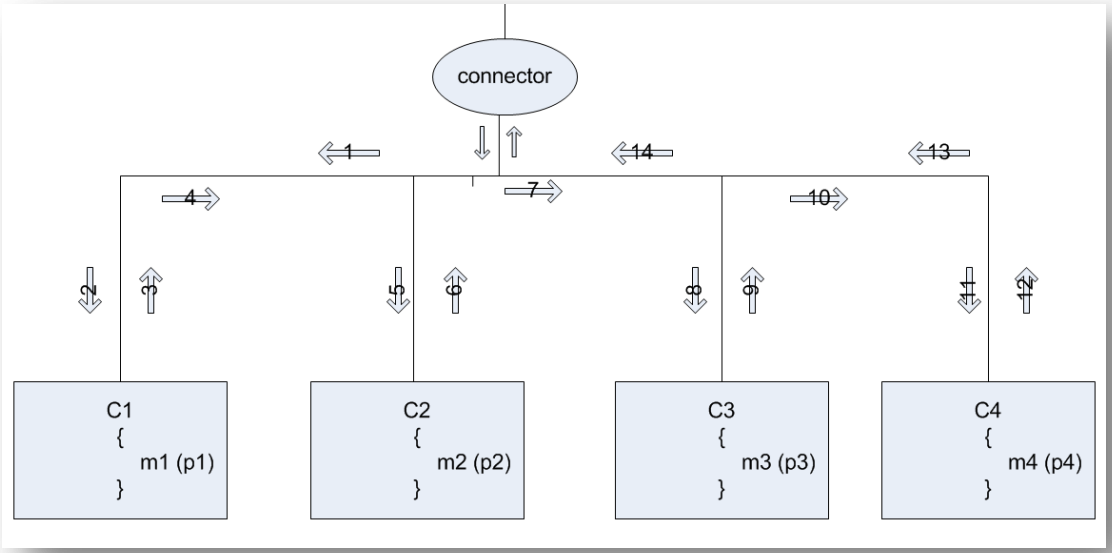

Figure 17: The general control flow

Note: The figure above is a general concept and it may not be applicable to all types of connectors. Some connectors like Selector only execute one of their sub-connectors depending on a condition being received.

The other operation specified in the 'ConnectorInterface.java' is executeConnector(). It is hard-coded in the 'Connector.java' and all other connectors inherit it. Unlike execute() method, this method will not be overridden, but it is called in inside each execute() method. This method is designed for execution of any method at any level of hierarchy. Having been called, it first identifies the type of the connector being sent as a parameter, and

then according to this type, it does some adjustment, like preparing arguments, so that the connector being sent can be executed. At the end, it calls the connector by invoking its execute() method. This iteration (calling the executeConnector() inside of each execute() method) will be carried out until it reaches to an invocation connector, resulting in invoking the actual operation in underlying computation unit.

To manage the control flow properly, connectors must be designed in a manner that standard control structures like sequencing, branching, and looping can be achieved. Among such connectors, I can refer to Sequencer and Pipe as of interest of this dissertation.

10.3.3.1. SEQUENCER

Figure 18 shows the UML diagram for Sequencer connector. It has its own implementation of execute method so that it functions according to abstract definition of the Sequencer connector.

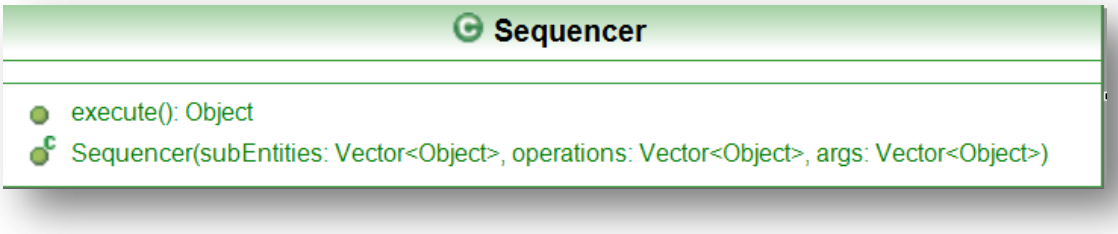

Figure 18: Sequencer.java

Sequencer is a kind of connector which represent sequencing control scheme. It essentially executes its sub-connectors in a predetermined sequence and returns any result produced by this set of sub-connectors. (Figure 19)

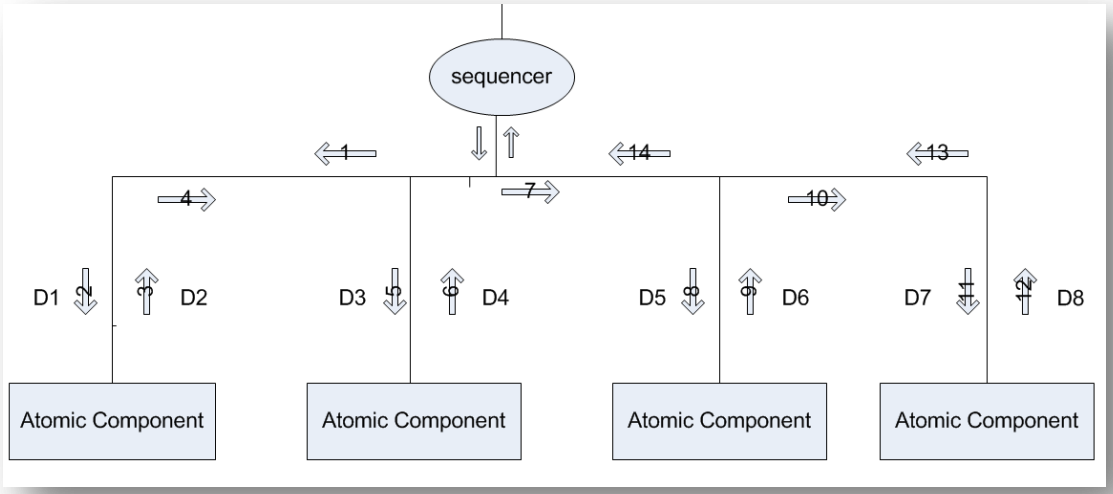

Figure 19: The control structure of sequencer

As it can be seen, each Sequencer connector has some sub-components attached to it. For the sake of simplicity imagine that these sub-components are atomic ones. The control initiates at top-level connector of each system. In this example, the top-level connector is sequencer itself. Having received the required parameters (D1, D3, D5, D7), the sequencer then starts execution of each of its sub-components and finally returns all the result back (D2, D3, D6, D8) to the next level up. It is important to note that the sub-components, atomic components in this example, do not call operations in the other components but they only receive required parameters, if any, execute the underling operation, and return the result. In other words, the sequencer handles the control flow between each of these atomic components.

To achieve this goal, there must be a design policy in place thereby the sequencer knows the order of execution of its sub components. The policy is very simple; the order is determined at time of instantiation of sequencer, that is, in the interface of this connector. (Figure 16)

10.3.3.2. PIPE

Figure 20 shows the UML diagram for Pipe connector. It has its own implementation of execute method so that it functions according to abstract definition of the Pipe connector.

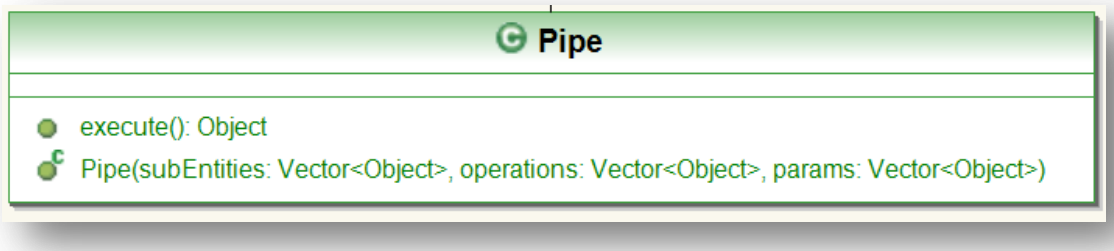

Figure 20: Pipe.java

Pipe is a kind of connector which represent sequencing control scheme along with piping any produced result from one component to another successively. When Pipe is instantiated, it is only supplied with the first parameters, D1, required for the execution of the first method and then other parameters are specified at run time during the execution of sub-components. (Figure 21)

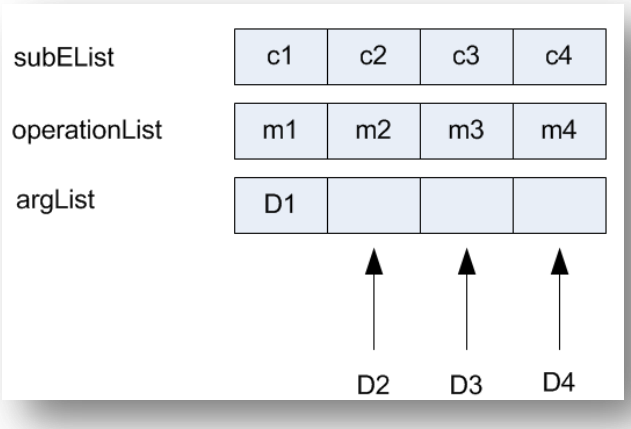

Figure 21: Data flow in Pipe

Pipe essentially executes its sub-connectors in a predetermined sequence and returns any result produced by this set of sub-connectors. (Figure 22) It should be clear from the figure below that, Pipe executes the first component with the given data (input parameter) D1, then gets the result back, D2, and sends it to the next component as its input parameter for execution of underlying method. In other words, Pipe only receives the first parameters from upper-layer components and then uses it for execution of first components. This process is done until all of the sub-connectors are executed and the final result is piped back to the next level up.

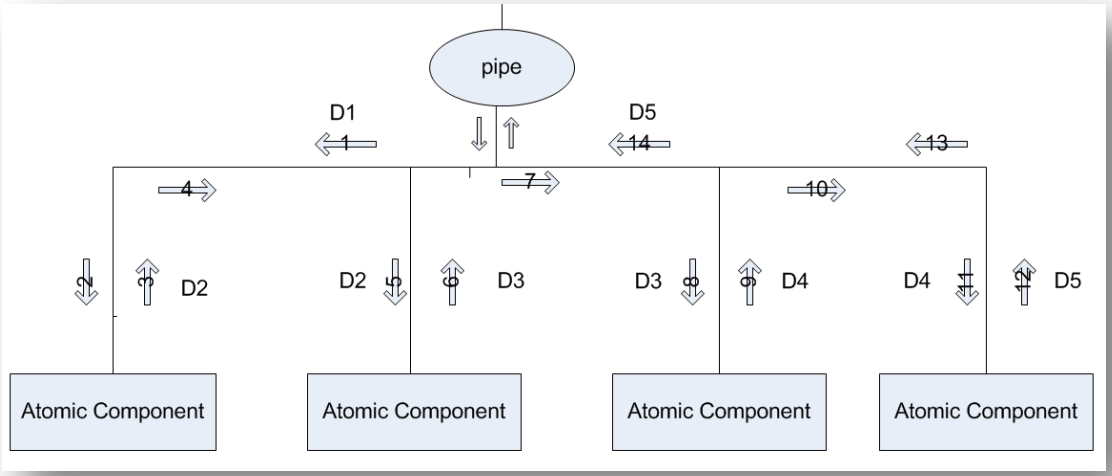

Figure 22: Control structure of pipe connector

10.3.3.3. INVOCATION

Figure 23 shows the UML diagram for Invocation connector. It has its own implementation of execute method so that it functions according to abstract definition of the Invocation connector.

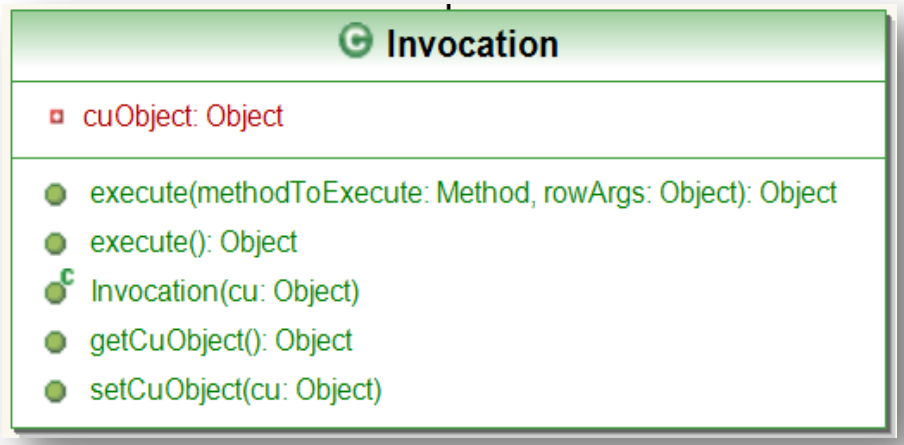

Figure 23: Invocation.java

The execute() method of Invocation connector is different from that of other basic connectors. As it can be seen in figure 23, the signature of execute method requires two parameters, one is of type Method, which is the underlying operation to be invoked by this invocation connector, and the other one is the parameter needed for execution of this method. Invocation connectors first convert the parameter received, rowArgs, to the right format, finalArgs, so that it can be used in invoke() method provided by java reflection mechanism. If invoke() method fails, then the 'result' would be assigned a value accordingly.

```
public class Invocation extends Connector
{
        ...
        result = methodToExecute.invoke( getCuObject(), finalArgs );
        ...
}
```

Figure 24:

10.3.4. COMPOSITE COMPOSITION OPERATORS

So far, we have seen the basic connectors and how they functions. Additionally, we have came to conclusion that basic connectors can be composed to form more complex control structure provide that resulting control structure has an intent behind the scene. This complicated control structure may aim at design patterns, that is, design patterns can be represented as a composite composition operator (composite connector). In such a design, patterns have concrete identity and thus, they can be reused.

In this dissertation, two well-known object-oriented design patterns, Observer and Chain of Responsibility are defined as composite connectors, ObserverCC and CoR respectively. The two resulting composite connectors are further composed together to form a new composite control structure named ObserverCoR.

10.3.4.1. OBSERVER COMPOSITION OPERATOR

Figure 25 shows the UML diagram for ObserverCC connector. It has its own implementation of execute method so that it functions according to abstract definition of the Observer design pattern.

Figure 25: ObserverCC.java

A composite connector is a composition of a set of basic and/or composite connectors. As it can be seen in figure 25, ObserverCC has two references to Pipe

and Sequencer as its data type in its source code. To be more explicit, the composite connector ObserverCC is defined in terms of Pipe and Sequencer basic connectors. (Figure 26) The Observer object-oriented design pattern is intended to define a one to many dependency between publishers and subscribers. Looking at the following diagram, we can understand that the Observer pattern can be represented as a combination of Pipe and Sequencer where the publisher is attached to the former and the set of subscribers is attached to the latter.

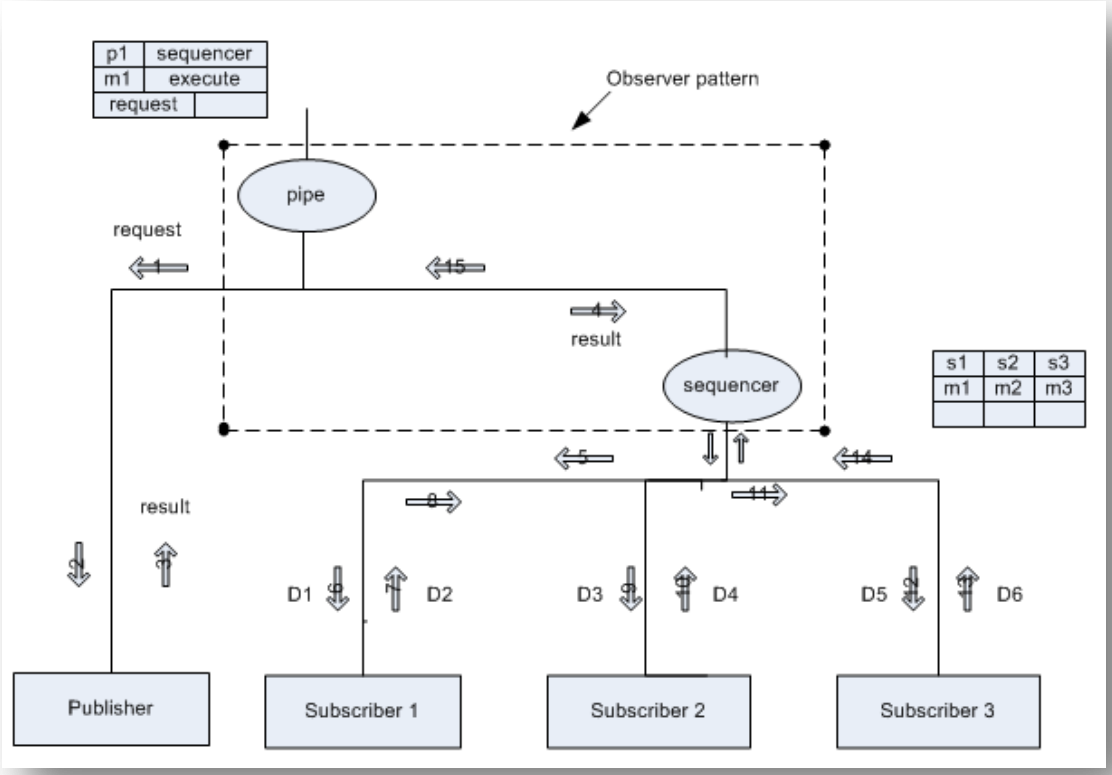

Figure 26: The control structure of Observer pattern

Pipe, as a top-level connector, initiates the control flow by sending data to publisher. Publisher receives it, changes its state, and then publishes the result. Next, the returning result is piped to sequencer as input parameter. As you can see, the value of sequencer's parameter is empty at the beginning and indeed returning result is copied there. Having been received, the parameter is unpacked

by the sequencer and it is distributed to its sub-components, i.e. the set of publishers. They finally react accordingly based on the valued being received. Of course, generally speaking publishers and subscribers need to match together, that is, they should be a dependency between them so that whenever the publisher updates, the subscribers are notified and react. Hence, valid participants need to be selected to be composed by the composite connector ObserverCC.

Unlike other basic connectors, the ObserverCC's constructor differs in terms of type and number of parameters. It receives five parameters and distributes parameters to its internal elements, i.e. pipe and sequencer. (Figure 27)

```
public ObserverCC( Vector< Object > publisher,
                   Vector< Object > publisherOpr,
                   Vector< Object > publisherArgs,
                   Vector< Object > subscribers,
                   Vector< Object > subscriberOperations )
    {
          this.pip = new Pipe( publisher,
                               publisherOpr,
                               publisherArgs );

          Vector< Object > vec = new Vector< Object >();
          this.seq = new Sequencer( subscribers,
                                    subscriberOperations,
                                    vec );
    }
```

Figure 27:

Later on, ObserverCC uses the references to pipe and sequencer to call their execute methods in execute method of itself. (Figure 28)

```
public Object execute()
	{

		subResult = this.pip.execute();
		...
		subResult = this.seq.execute();
		...
		return result ;
	}
```

Figure 28:

10.3.4.2. CHAIN OF RESPOSIBILITY COMPOSITION OPERATOR

Figure 29 shows the UML diagram for CoR connector. It has its own implementation of execute() method so that it functions according to abstract definition of the CoR design pattern.

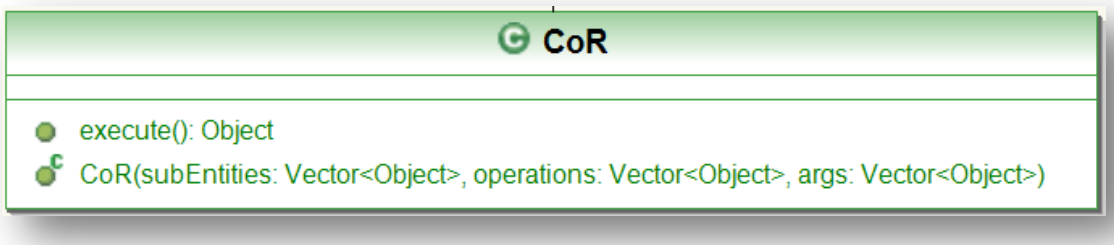

Figure 29:CoR.java

Not only composite connectors, but also basic connectors can represent design patterns. The chain of responsibility pattern defines a chain of handlers who receive a common request by a sender and as soon as the request is handled by first node within the chain, the result is returned back to the sender. This control scheme pretty much resembles to that of sequencer since in the sequencer also a parameter (request) is passed to a set of components (handlers) successively. (Figure 30)

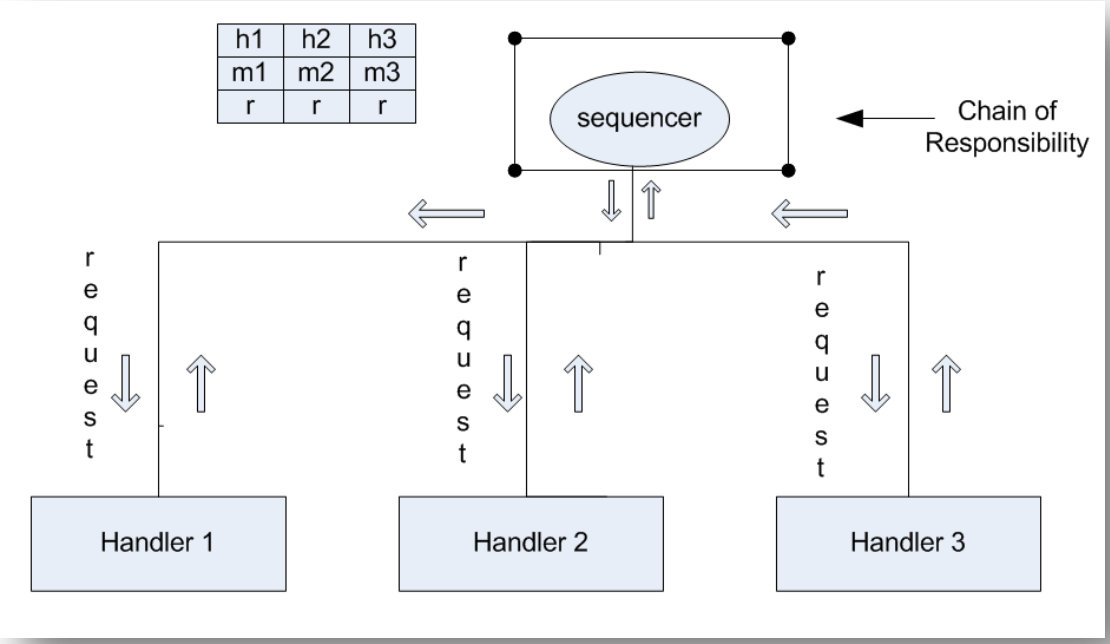

Figure 30: Chain of responsibility pattern

As it must be clear by now, a sequencer naturally executes each of its sub-components (they are assumed to be atomic components) successively, thus, In order to achieve the control structure of the Chain of Responsibility pattern, there must be some design policy such that as soon as the request is handled, the control flow returns back. The following excerpt (Figure 31) demonstrates such policy, indicating that if the result being received from a handler (subResult) is not of type Exception, therefore it has succeeded and iteration must be stopped. Indeed, the indicated if() structure is inside of a for() loop that controls the whole iteration between sub-components, thus the break; terminates the iteration and flow of control leaves the loop.

```
if ( !( subResult[ i ] instanceof Exception ) )
{
        succeedR[ 0 ] = subResult[ i ];
        break;
}
```

Figure 31:

10.3.4.3. OBSERVER-COR COMPOSITION OPERATOR

So far, we have seen how to define the design patterns as composite connectors so that their reusability is truly achieved in practice. At this point, I would like to point out that how composite components can be composed together to form a new composite structure. I explain the composition of ObserverCC and CoR, which results in ObserverCoR composite connector.

Note: Please note that in object oriented programming language, there is no such a pattern named ObserverCoR. It is merely defined in this dissertation to demonstrate the composition of composite connectors.

Figure 32 indicates an outline of the ObserverCoR composite connector. ObserverCoR composite connector receives a request and passes it to a set of

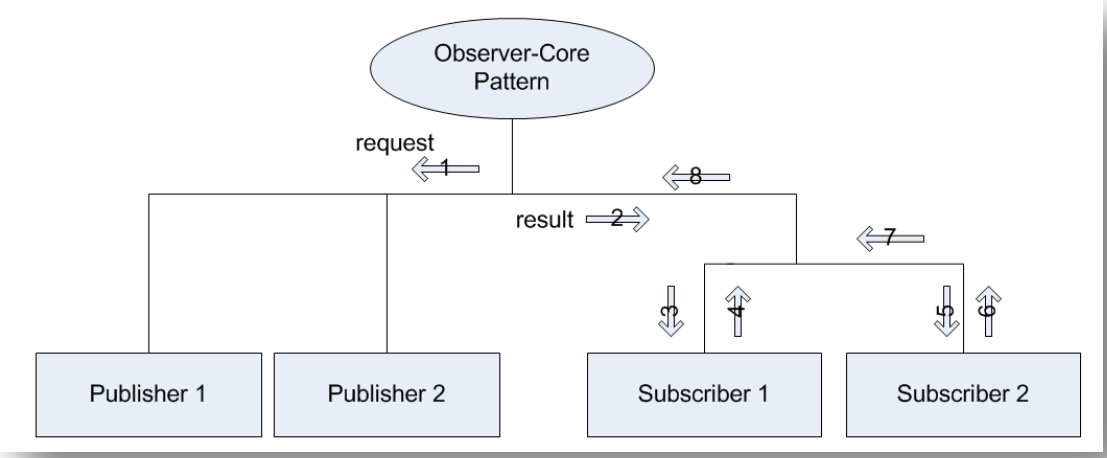

Figure 32: Observer-Cor pattern

handlers. After the request is handled by the first handler, the result is sent to subscribers to notify them. The handlers compose the chain (in Chain of Responsibility pattern) and play the role of publishers (in Observer pattern). To be more specific, by combining Observer and COR patterns together, we make it possible to have more than one publisher. At each time, when a request arrives, depending on the type of the request on of these publishers is executed and then the output will be sent to all subscribers regardless of which publisher (handler) has already been executed. This implies that

- Firstly, the output of all publisher must be of the same type
- Secondly, this output (type of output) needs to be matched to input of all subscribers. All subscribers also have the same type of input.

These criteria enable ObserverCoR to pipe the output generated by any of publishers to all subscribers. Figure 33 shows the UML diagram for ObserverCoR connector. It has its own implementation of execute() method.

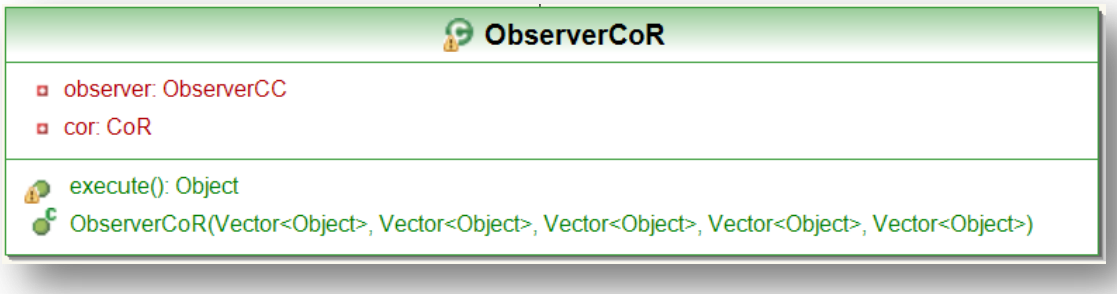

Figure 33: ObserverCoR.java

The ObserverCoR composite connector is composed of two composite composition operators, namely ObserverCC and CoR. (Figure 35) To enforce this in practice, its constructor is redefined with five parameters for initialization of its internal elements, i.e. observer and cor. (Figure 34)

```
public ObserverCoR( Vector< Object > publisher,
                    Vector< Object > publisherOpr,
                    Vector< Object > publisherArgs,
                    Vector< Object > subscribers,
                    Vector< Object > subscriberOperations )
    {

          Vector< Object > vec = new Vector< Object >();
          this.cor = new CoR( publisher, publisherOpr, vec );

          Vector< Object > subConn = new Vector< Object >();
          subConn.add( cor );

          Vector< Object > subOper = new Vector< Object >();
          subOper.add( "execute" );

          this.observer = new ObserverCC( subConn, subOper,
                                          publisherArgs,
                                        subscribers,
                                        subscriberOperations );

    }
```

Figure 34:

If you notice to the figure 35, you will find out that the parameters for seq2 is empty at the time of initialization. It is indeed the output that will be generated by one of the publishers in the chain. This empty value corresponds to publisherArgs in the signature of ObserverCoR constructor in figure 34.

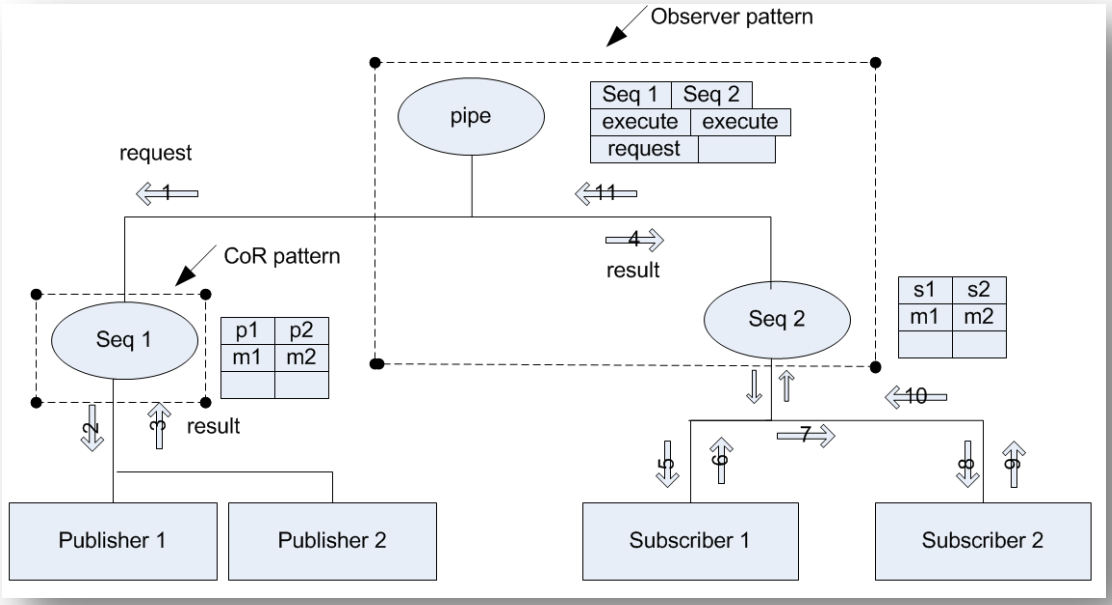

Figure 35: Observer-Cor pattern control flow structure

10.3.5. A PLAIN IDE AND A SAMPLE SCENARIO

Up until now, we have witnessed all of the key points of this dissertation. Now, I like to attract your attention to a plain IDE (Integrated Development Environment) that brings all discussions we have had in previous chapters in one place. In other words, it is implementation phase of the solution offered earlier. The solution, using design patterns in components-based software development, is discussed in terms of theory in preceding sections and now this simple IDE brings the solution into action. There would a sample scenario and components that represents such a scenario.

The IDE is designed in java language with three packages, namely components, connectors, and project. (Figure 36) The connector package contains almost all of the classes discussed before, like Connector.java, Pipe.java, and so on. The component package contains all sample computation units as well as AtomicComponent.java class which was discussed earlier. Last but not least, the project package just contains one class named MainWindwo.java, which forms

the GUI (Graphical User Interface) of the IDE. It therefore needs to import the two other packages.

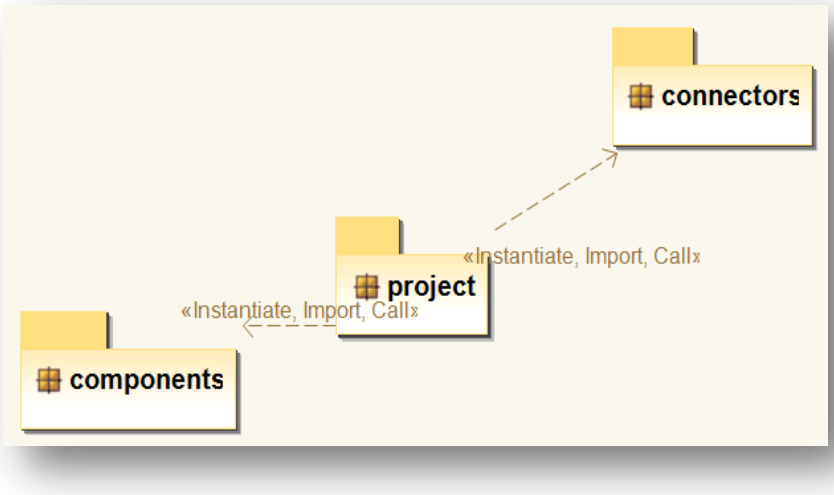

Figure 36: The whole system hierarchy

Figure 37 shows the UML diagram of MainWindow.java that constructs all windows comes up during the execution of the system. It is written using swing package of java language and complementary comments are given in the source code. The IDE basically provides the services of

- Allowing users to choose the computation units file from a repository, which is essentially a pre-defined directory
- Choosing the desired method from computation units so as to be invoked by its invocation connector
- Giving the atomic and composite components user-generated names so that they can be used later on during the execution
- Selecting components and composing them together
- Run the final composite connector and see the result

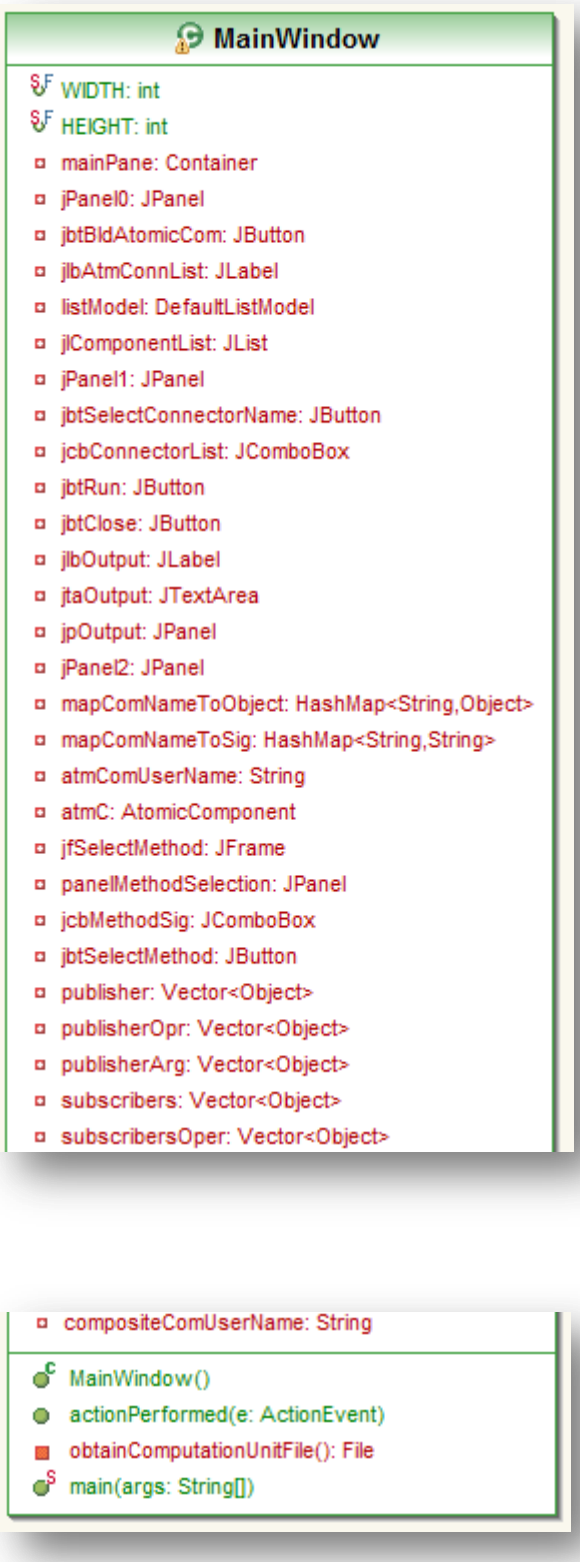

Figure 37: MainWindow.java

10.3.5.1. SAMPLE SCENARIO

The sample scenario is that we have a set of manufactures that form the publisher and a set of retailers and wholesalers that form the subscribers. Whenever the row material becomes available, the manufacturers build a product and then dispatch it to the publishers. Depending on the type of the row material being supplied, at each time one and only of the manufacture uses this row material and build a new product. At our example, the car manufacture is the target. It is notified by a message that the row material is available, and then it builds a new car and passes this product as a message to all subscribers so that they can purchase it.

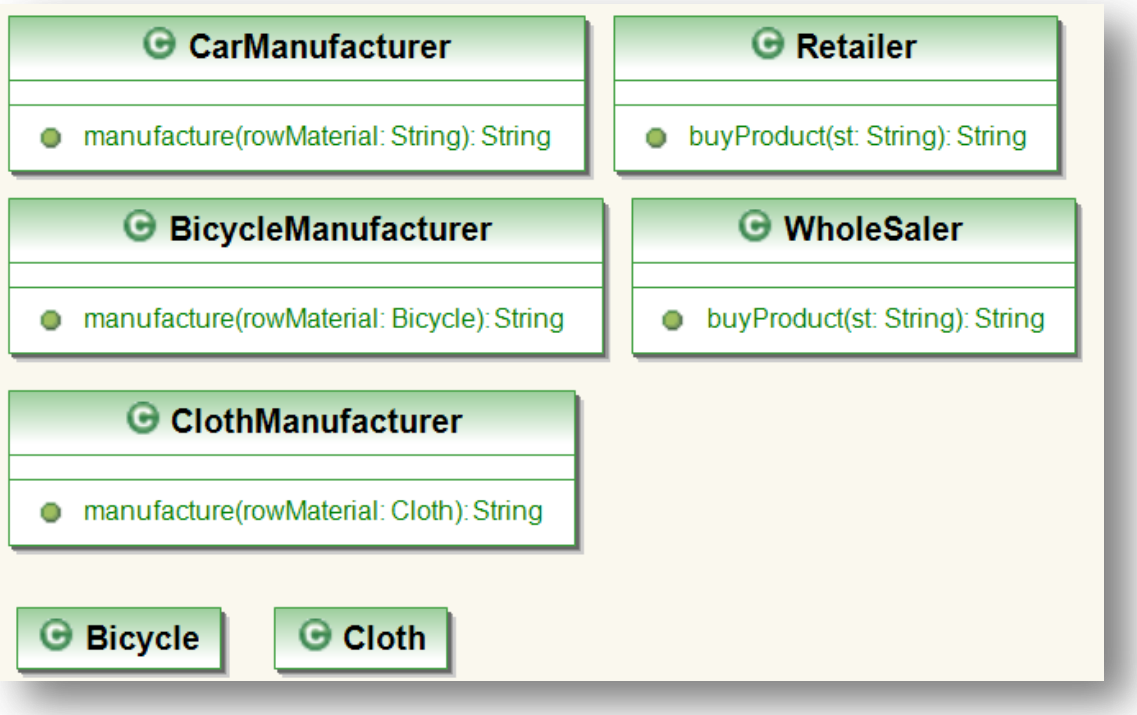

Figure 38: Computation units

Here is the computation units used in the sample scenario:

- CarManufacturer.java
- BicycleManufacturer.java

- ClothManufacturer.java
- Retailer.java
- Wholesaler.java

10.3.5.2. SNAPSHOT OF THE SYSTEM IN ACTION

At this stage, I take you through the snapshots of different stages of running the IDE. I demonstrate how to use the IDE and that how sample scenario works.

- This is basically the very first window that comes up. It tells you how to use the IDE and also reminds you of cares that need to be taken when using the IDE and composing components. It contains some useful guidelines.

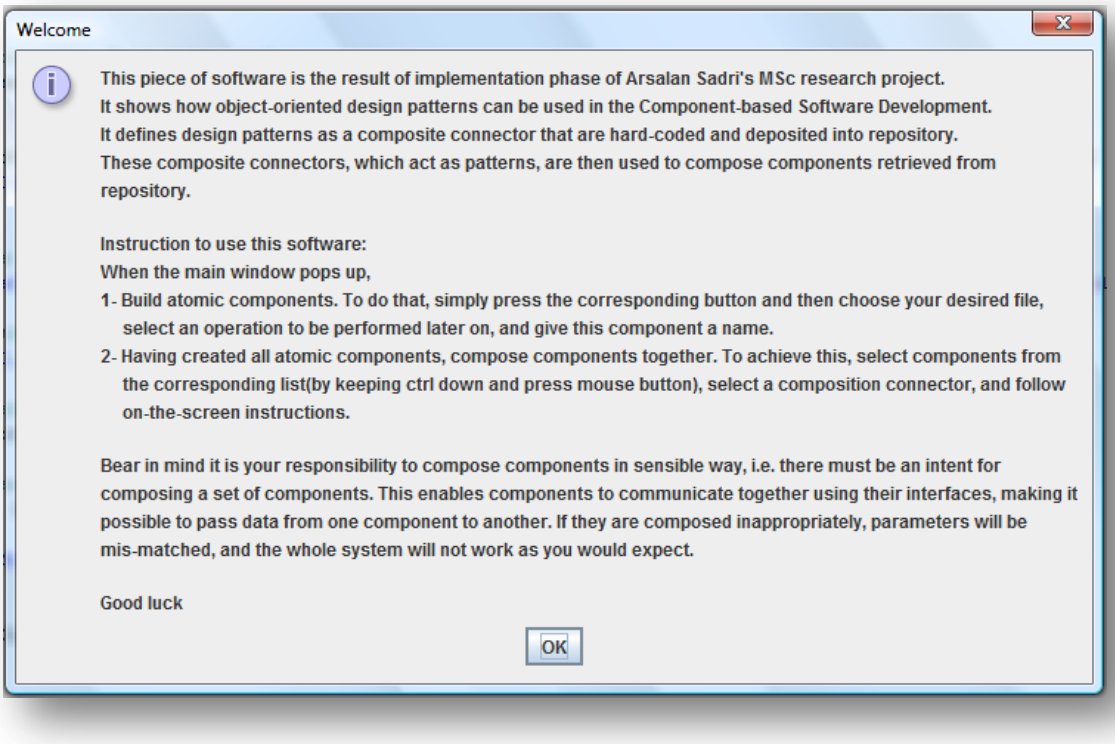

Figure 39: Guiding window

- The following window indicates how to deploy the sample scenario using computation units provided for this scenario.

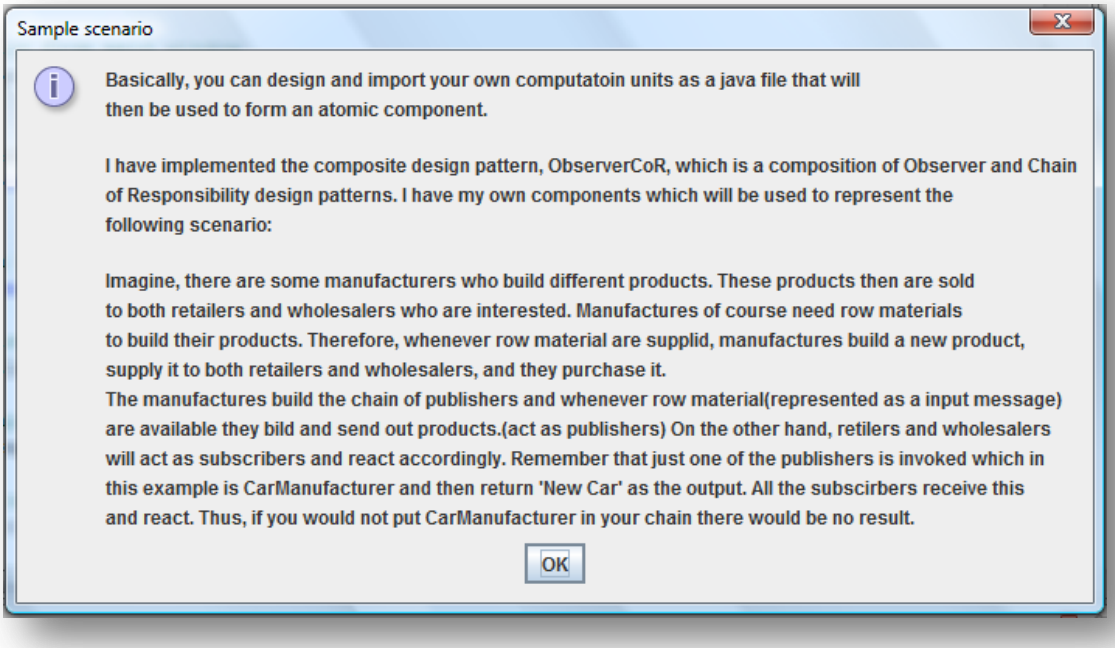

Figure 40:

- The following window is the main window of the program. It includes a button for building atomic components on the left-hand side. Having been created, the components will appear on the component list, titled with <- - Components- - >.

Having created all the atomic components, we then need to build our composite ones. To accomplish this, we simply first select all components (in sample scenario) and then select the Observer-CoR composition operator from the combo box, and finally press the corresponding button for building the composite components. After the final composite component is created, it will appear in the list. At this point, all we do is just selecting it and press Run button. Supply the right parameter, press okay and the output will be there in the output pane.

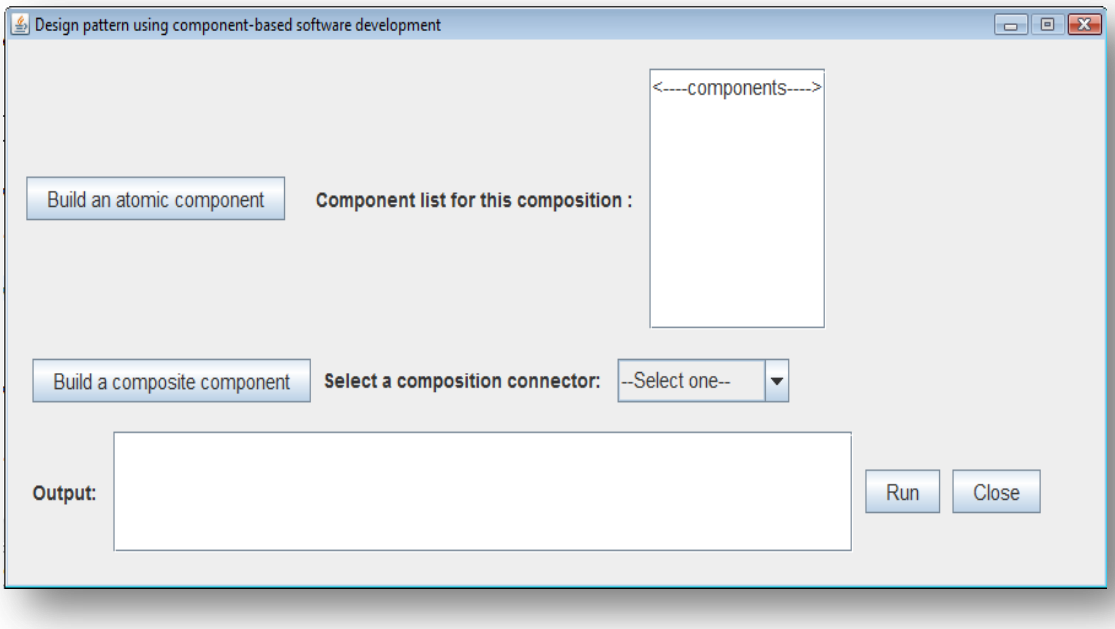

Figure 41:

- Selecting the computation units from the repository.

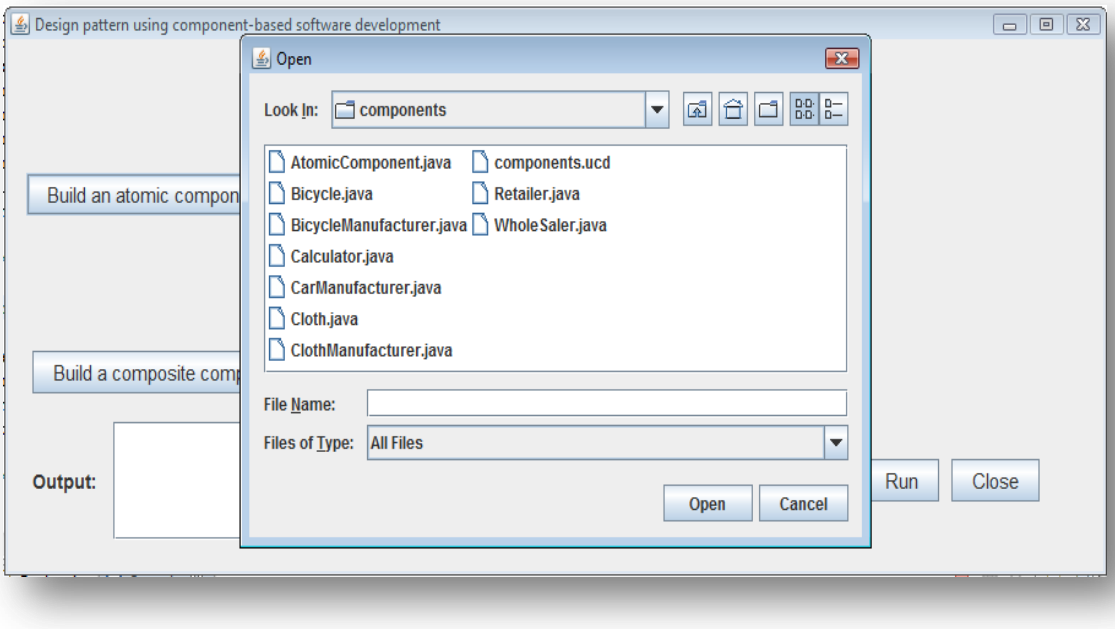

Figure 42:

- Giving it a name.

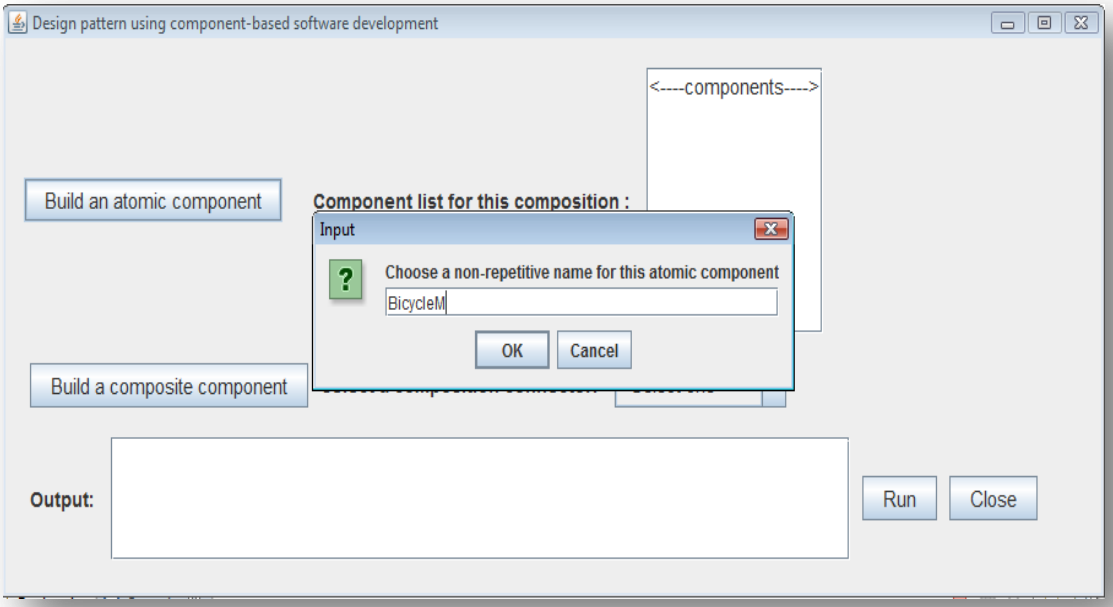

Figure 43:

- Choosing one method from the list of operations of the computation unit

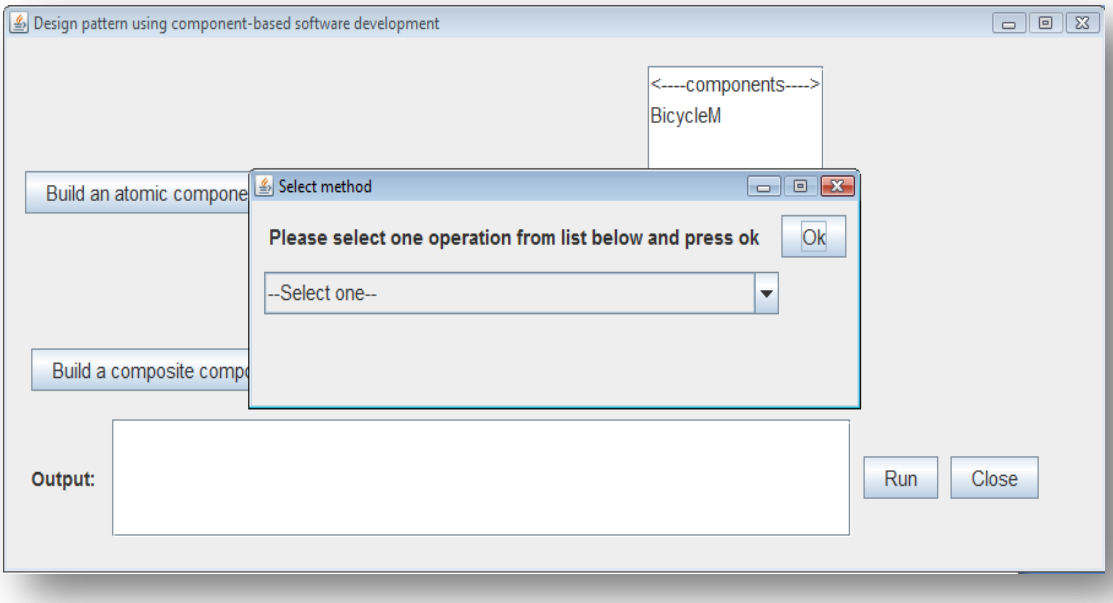

Figure 44:

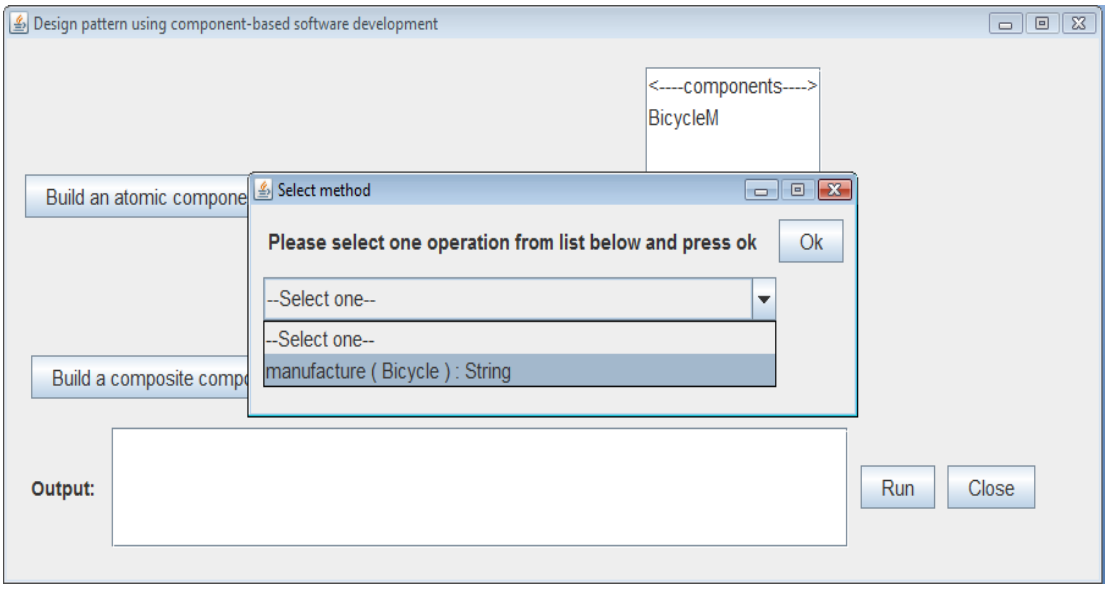

Figure 45:

- As you see, the new atomic component is now in the list

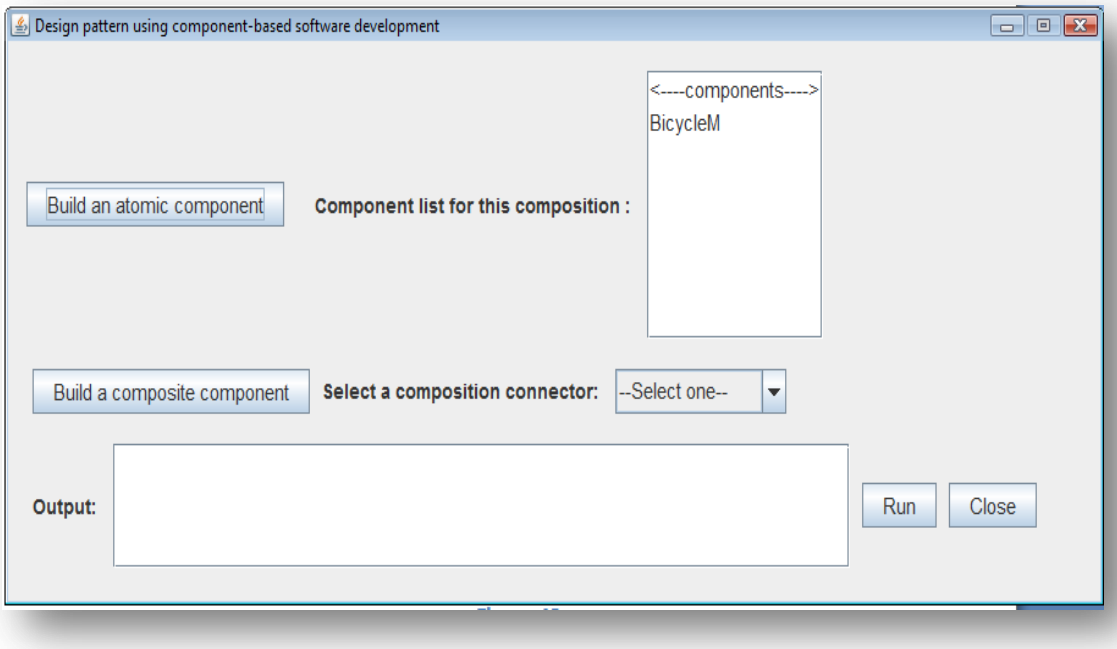

Figure 46:

- Selecting another computation unit and giving it a name

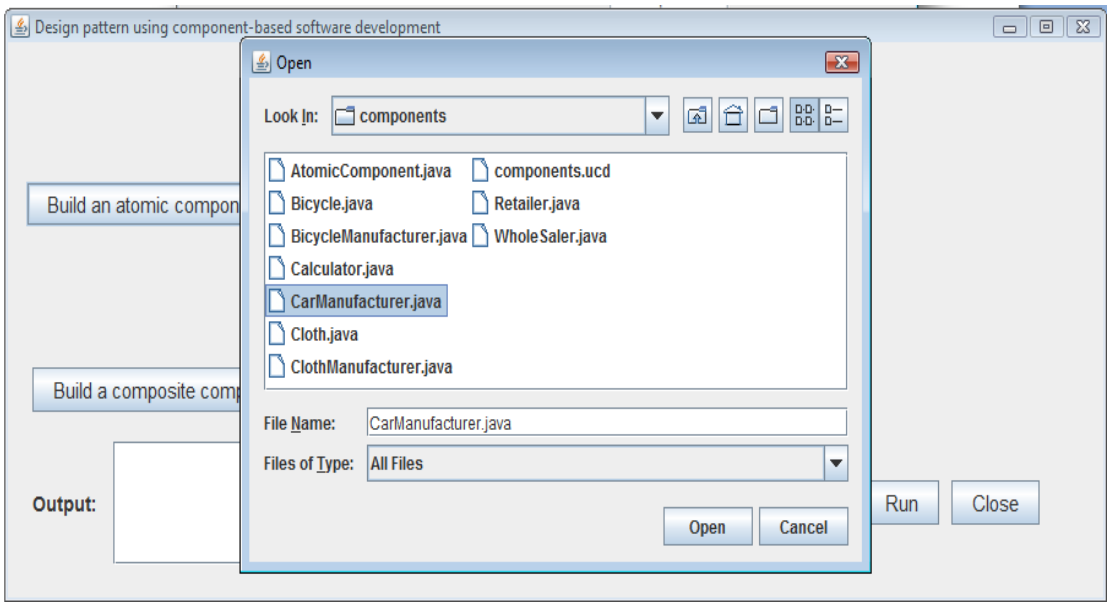

Figure 47:

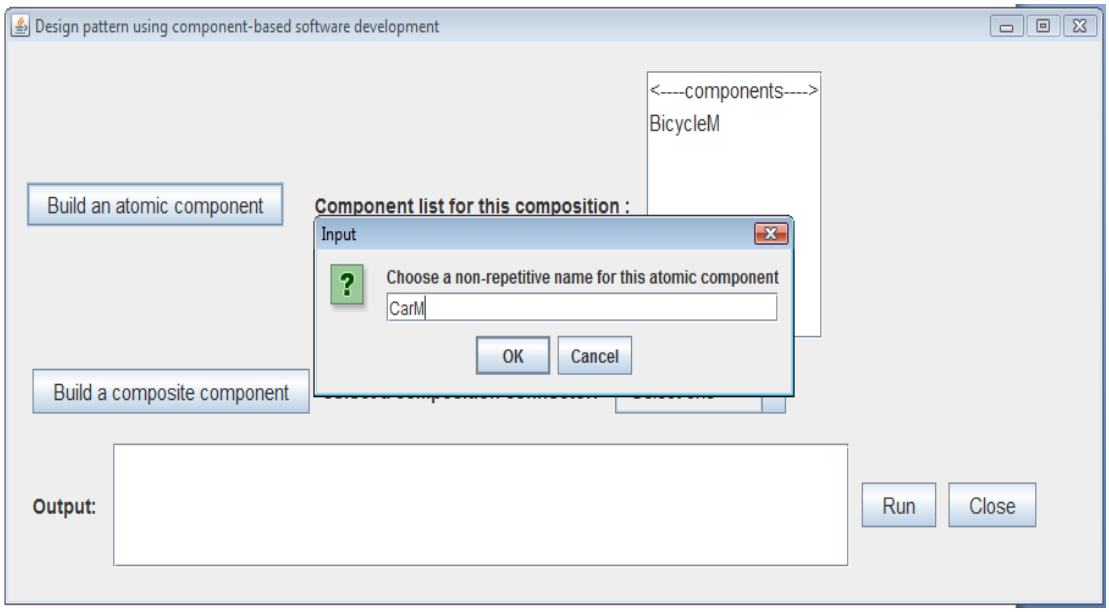

Figure 48:

- Choosing the method

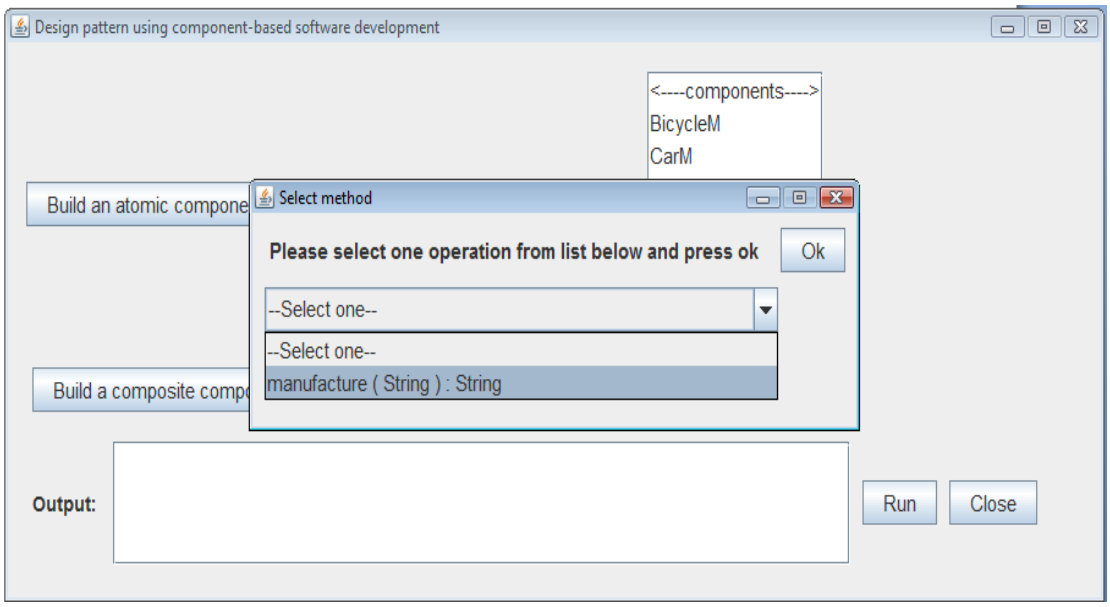

Figure 49:

- Selecting another computation unit and giving it a name

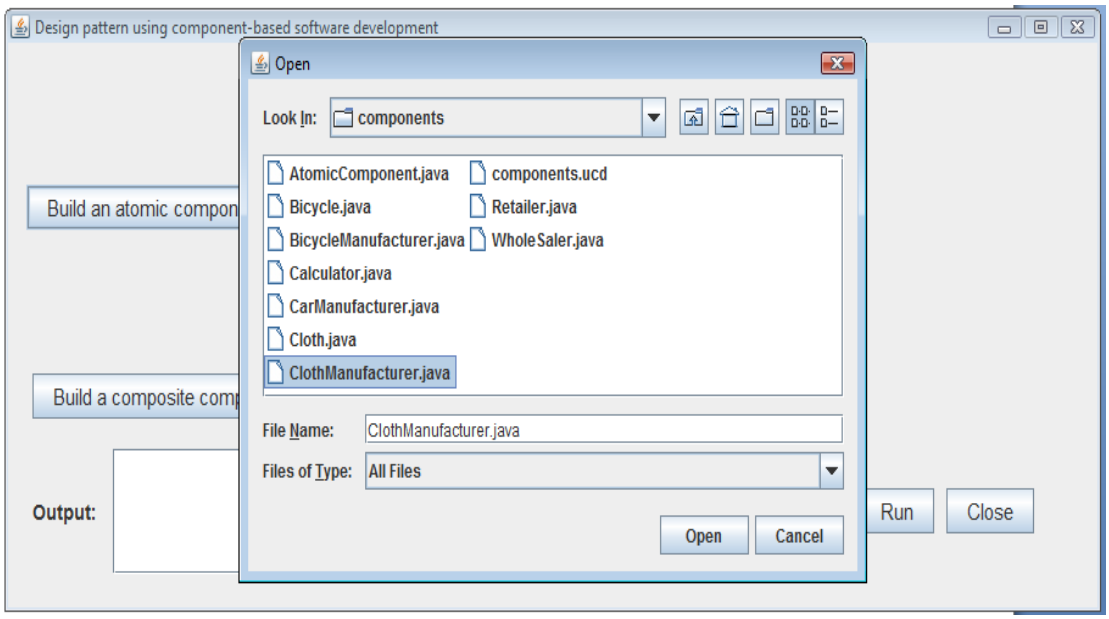

Figure 50:

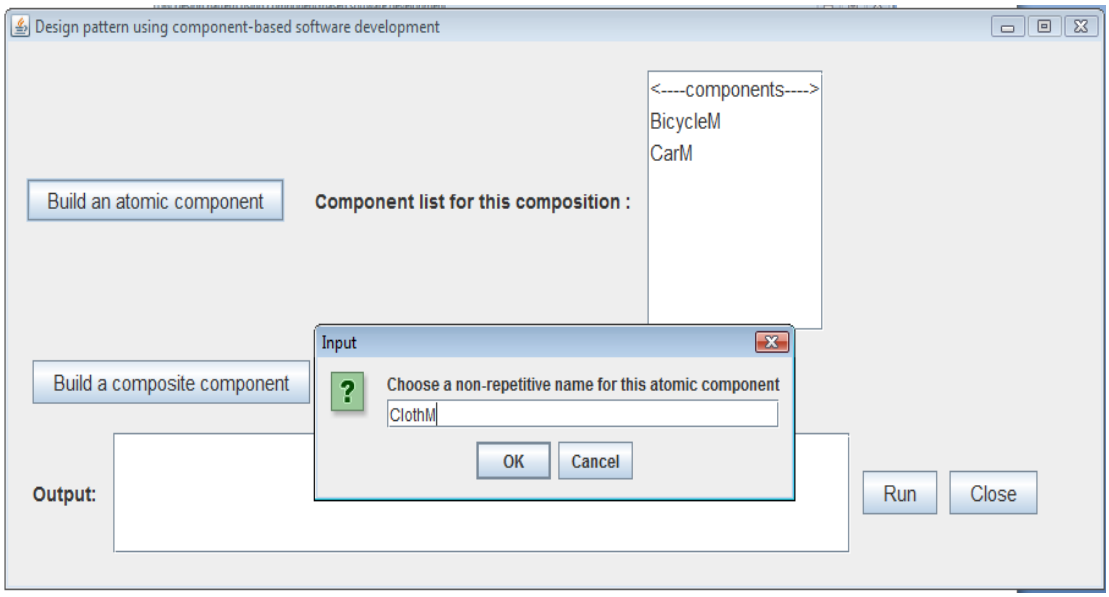

Figure 51:

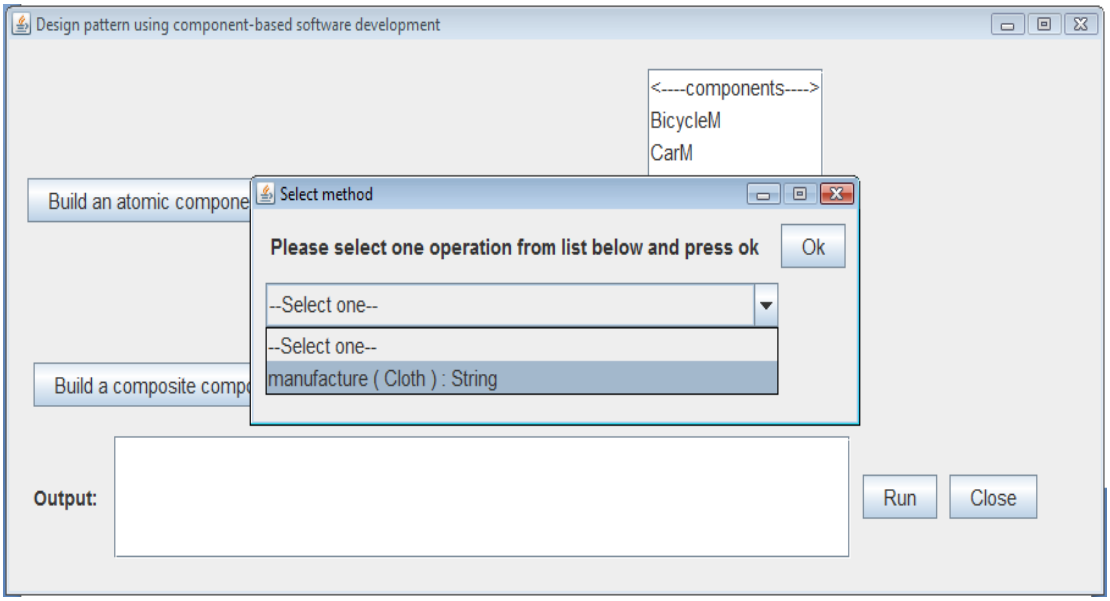

Figure 52:

- So far, we have selected all our publishers. Now we want to select subscribers which are retailers and wholesaler. Any numbers of these two atomic components can be created. For simplicity, we now create just one instance of each.

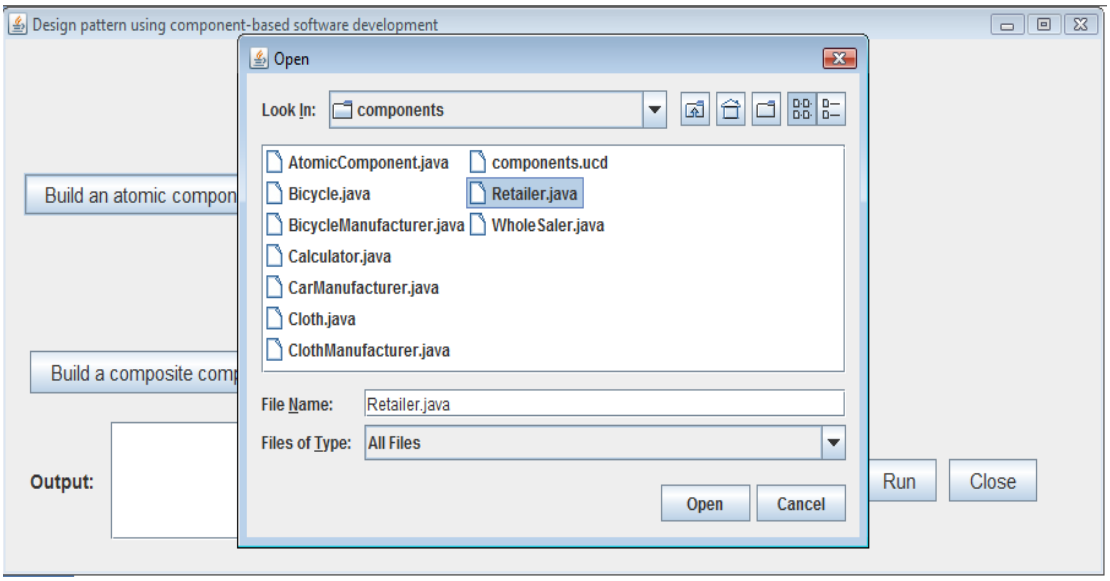

Figure 53:

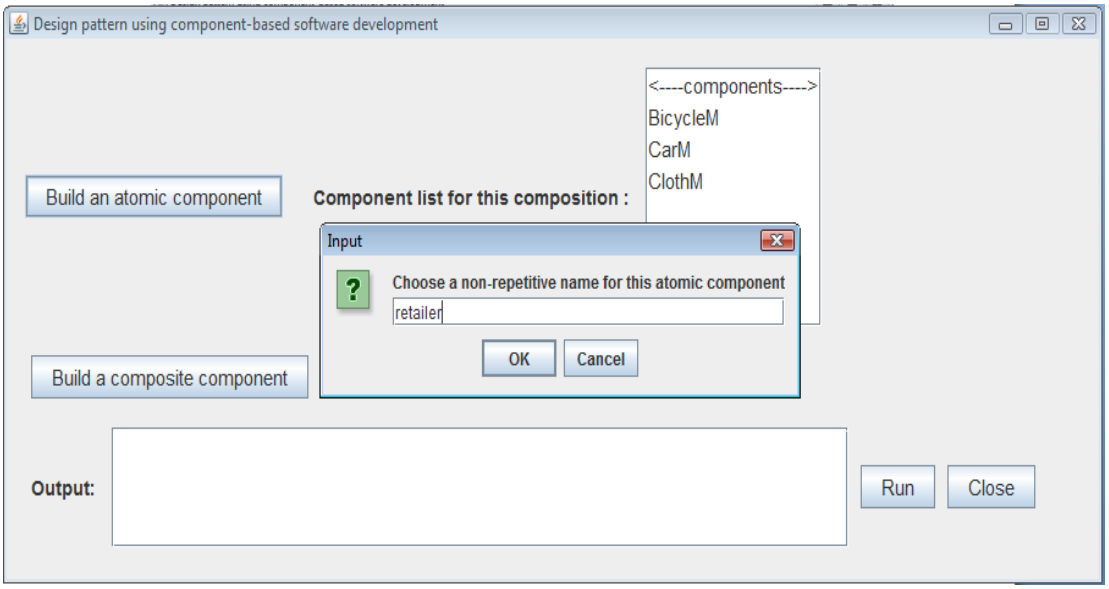

Figure 54:

- Choosing the buyProduct() method of subscribers so that whenever the first publisher publishes the new product, the subscribers can be notified any buy the corresponding product.

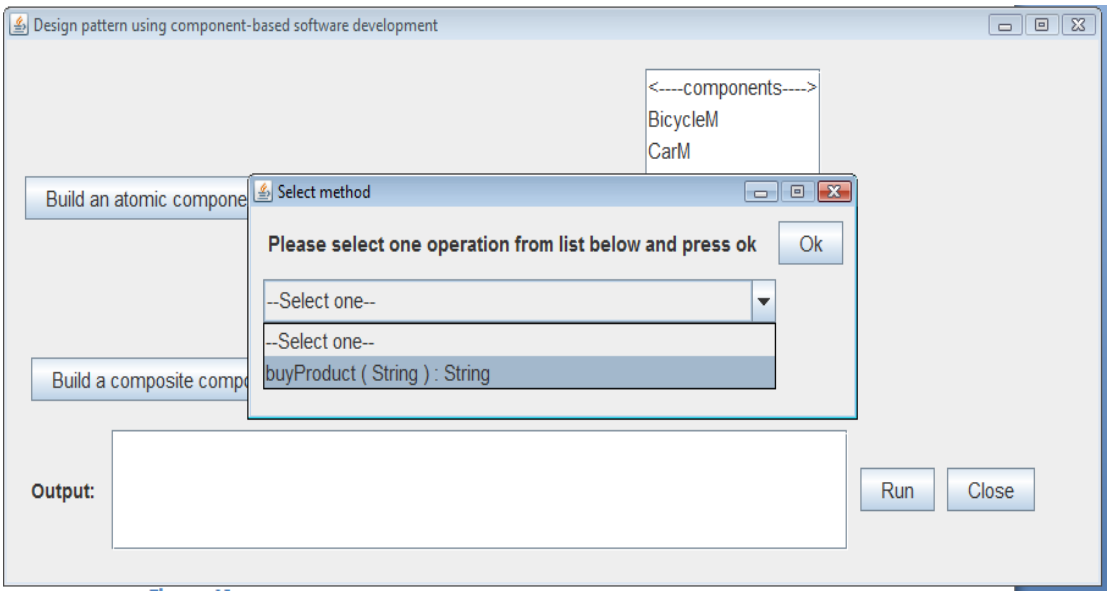

Figure 55:

- Choose the wholesaler as the second subscriber

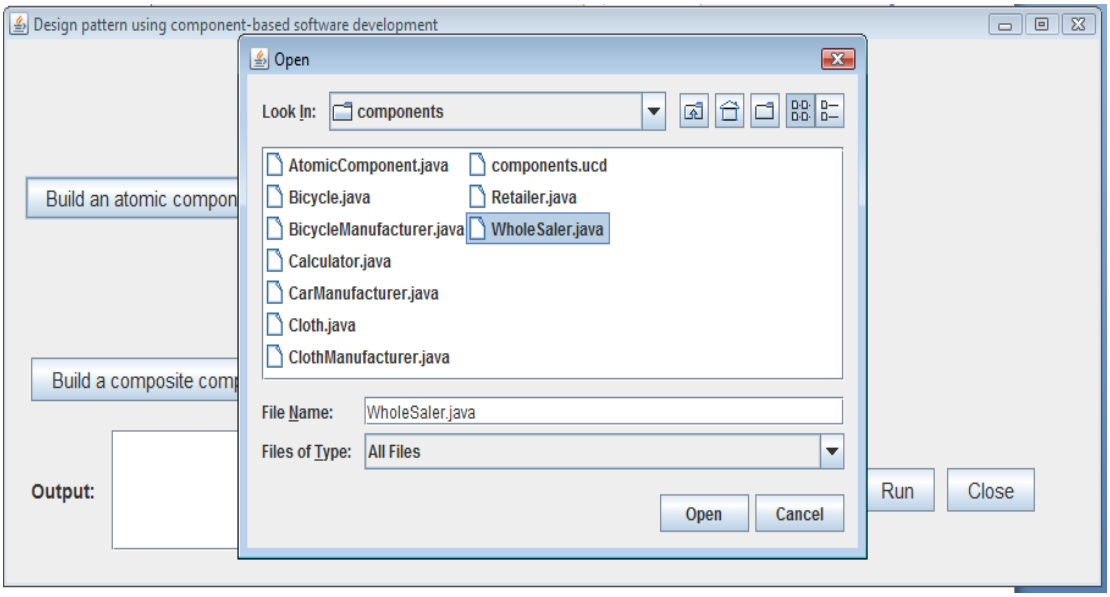

Figure 56:

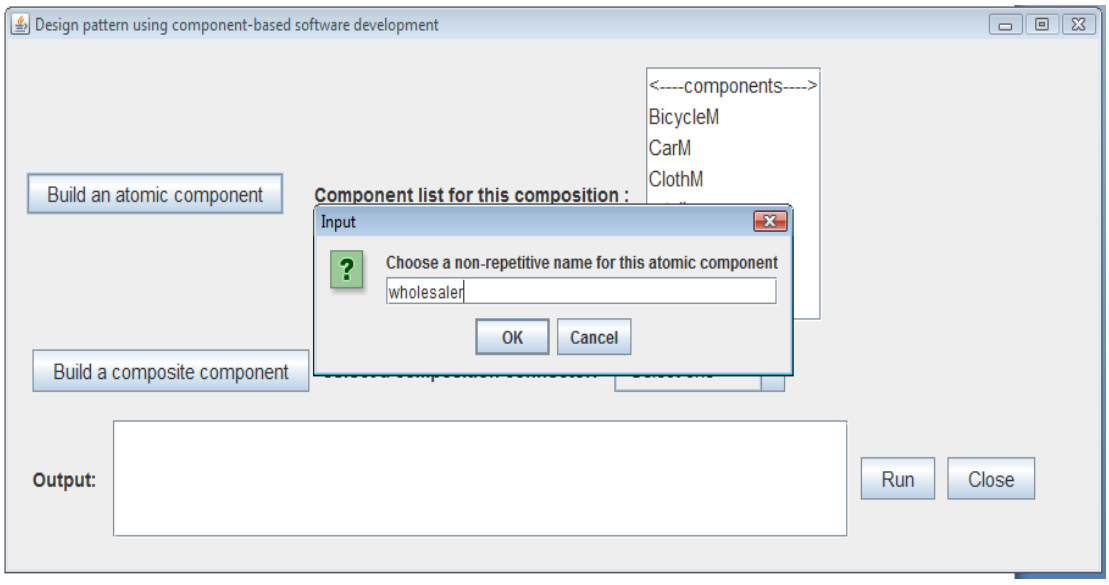

Figure 57:

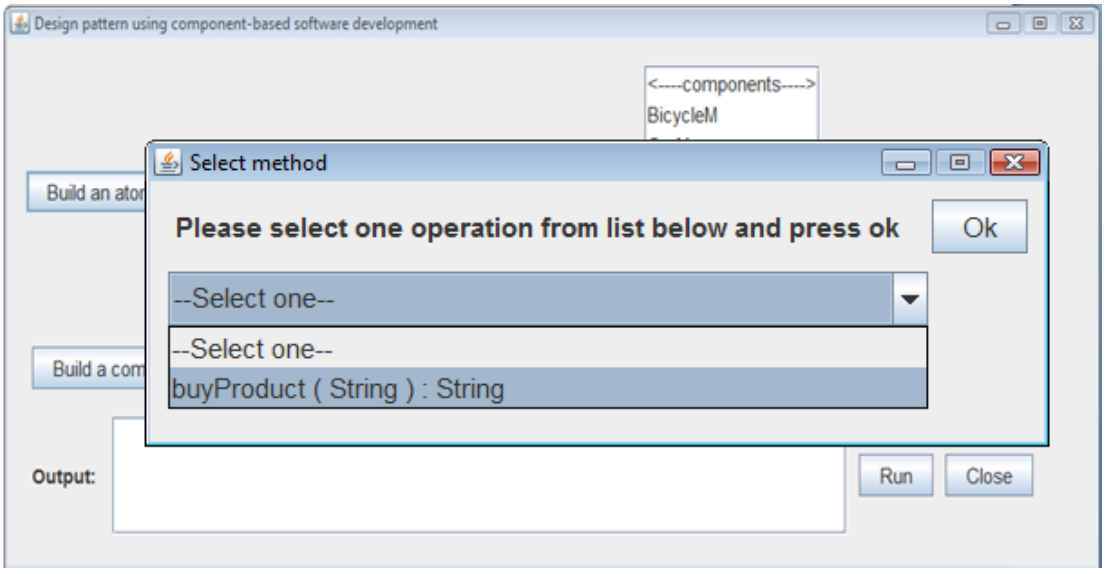

Figure 58:

- By now, we have built all of our atomic components. At this stage, as it can be seen, we select all atomic components as well as Observer-Cor composition operator, and then press build composite component button.

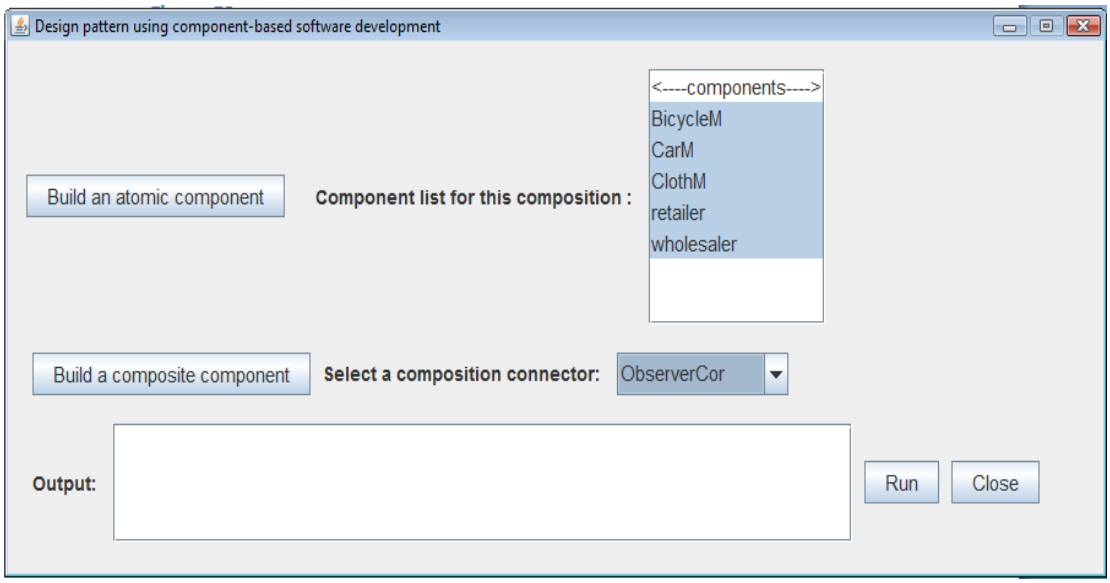

Figure 59:

- Giving it a name

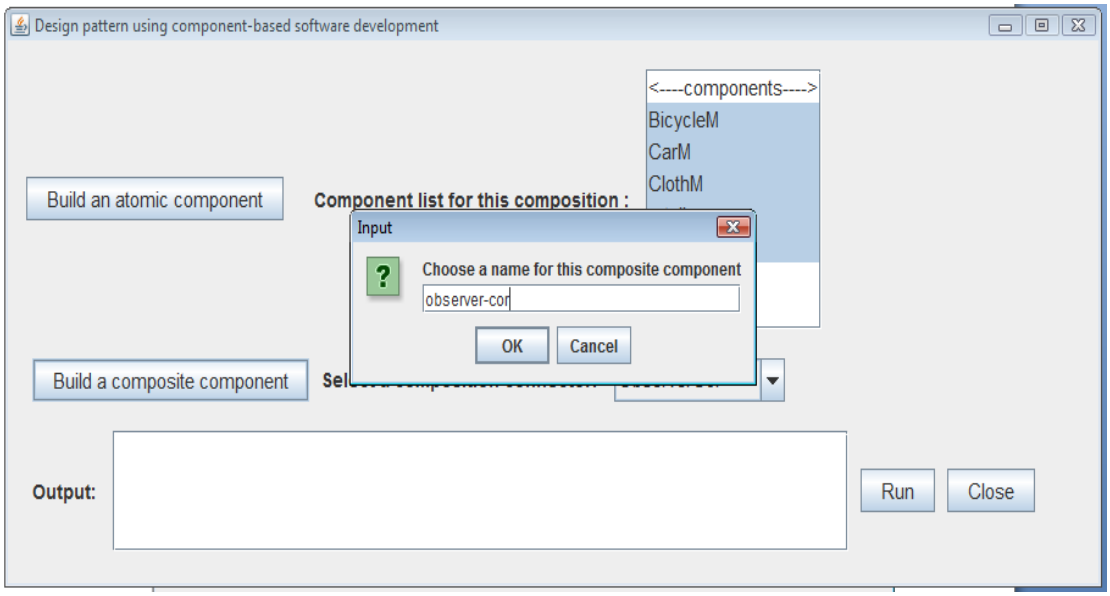

Figure 60:

- Let the system know which atomic component plays the role of publisher and which one plays the role of subscriber.

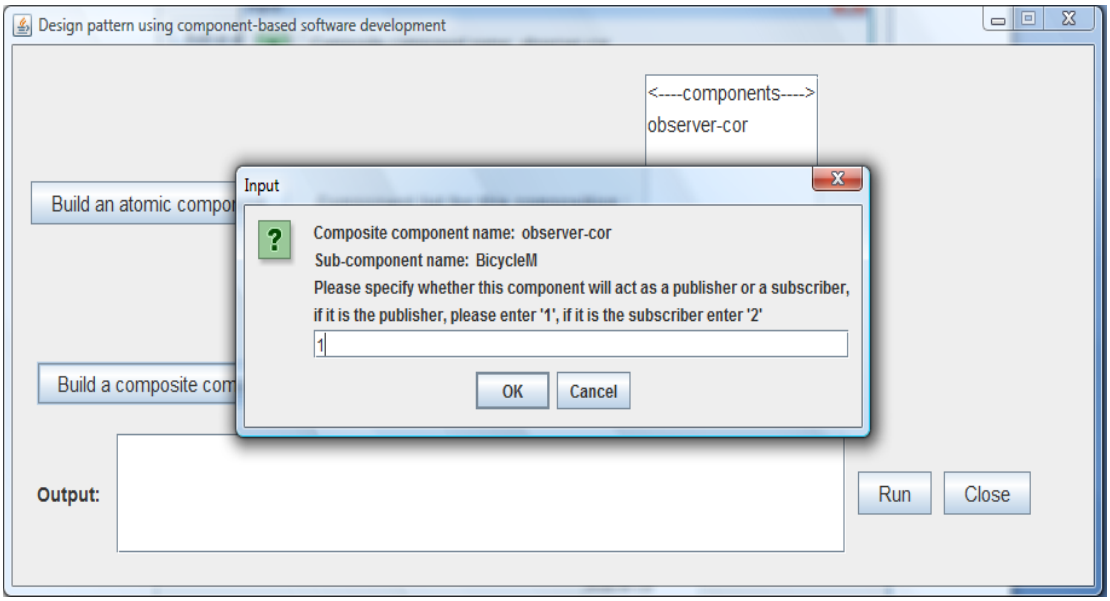

Figure 61:

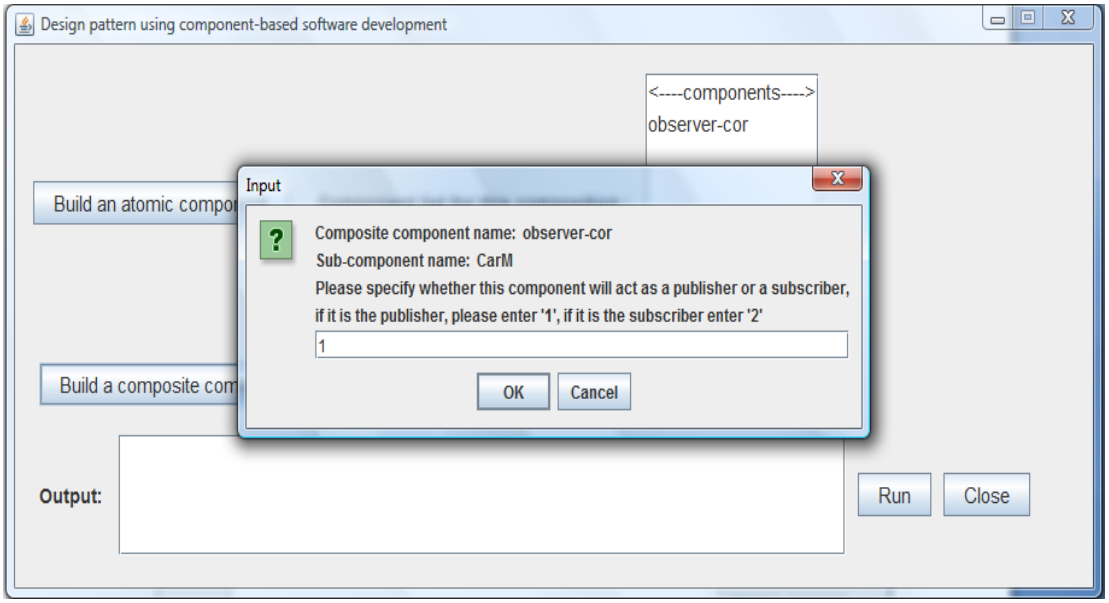

Figure 62:

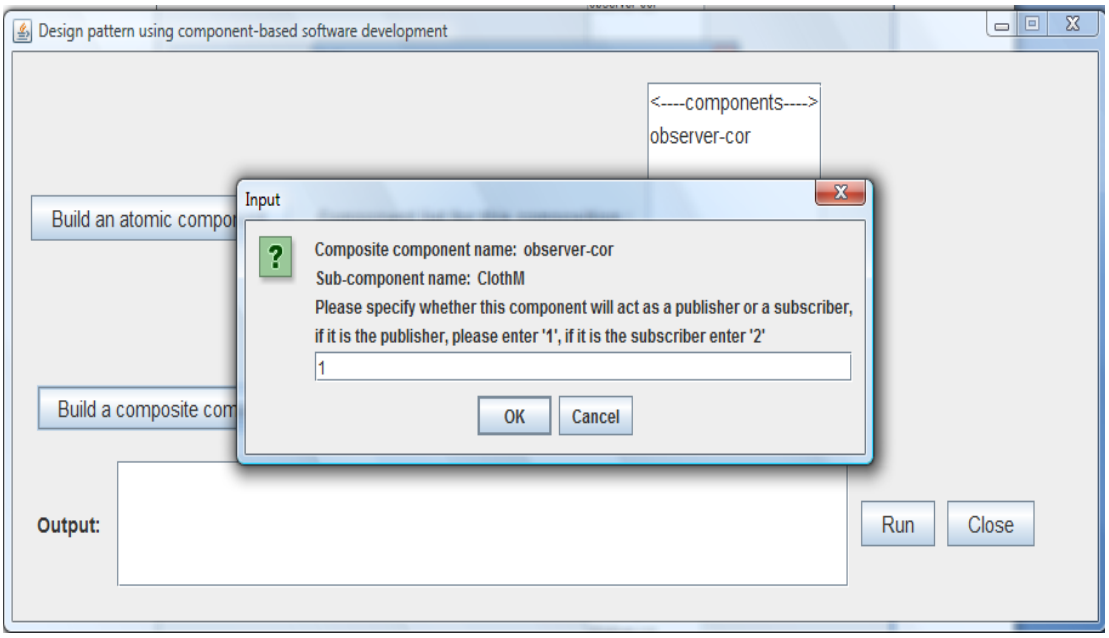

Figure 63:

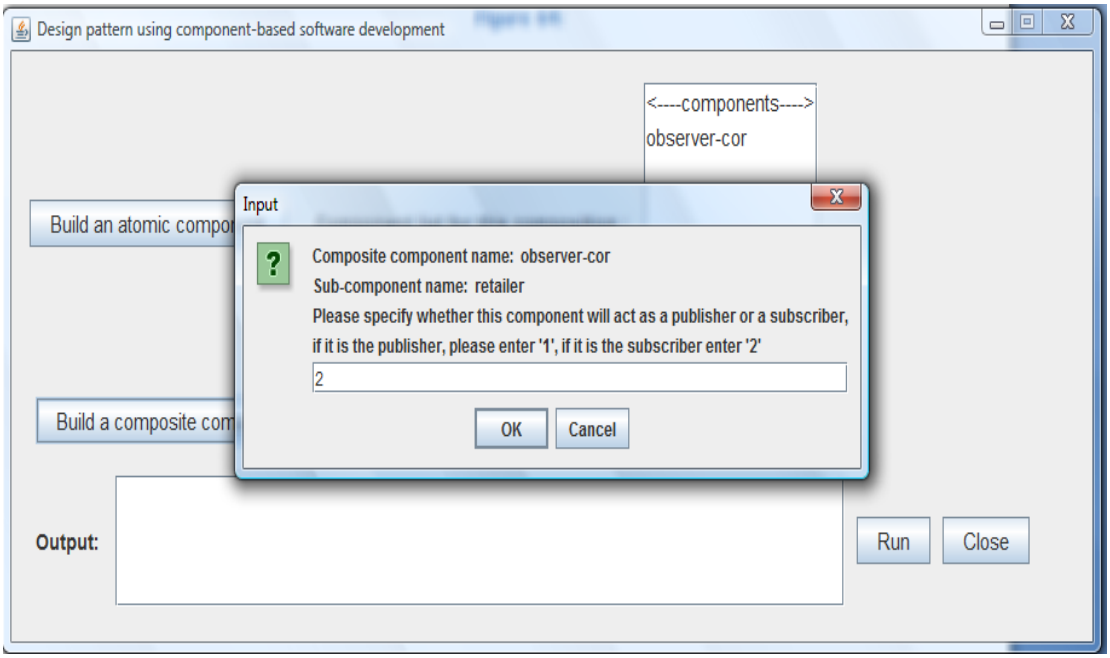

Figure 64:

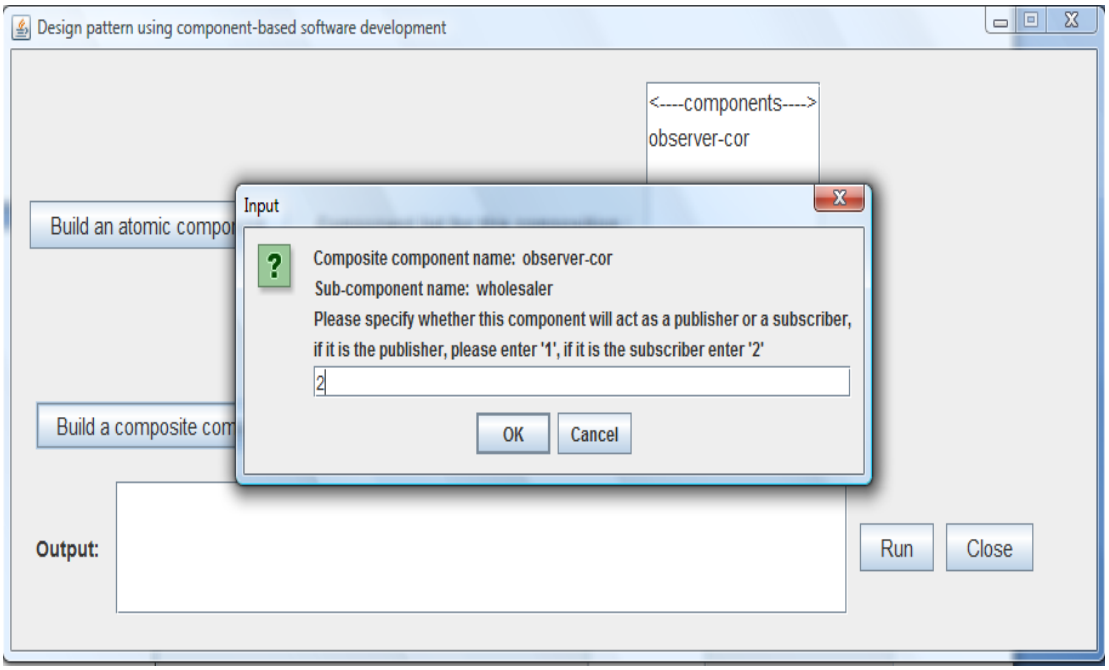

Figure 65:

- Selecting the final composite component and then press Run button

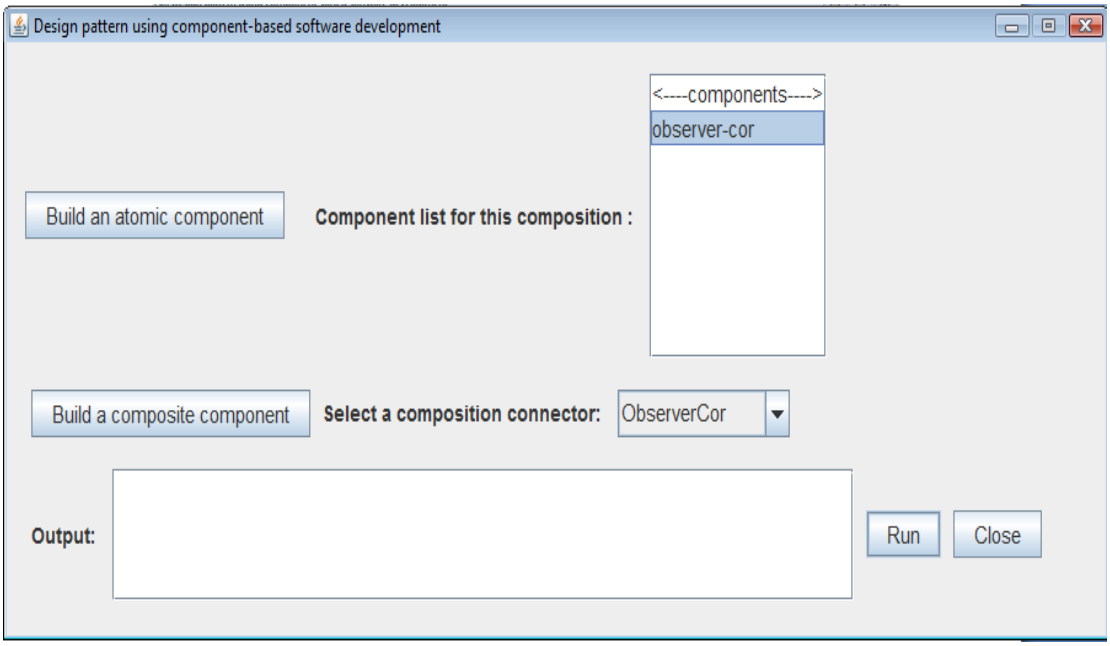

Figure 66:

- Putting the input as a message saying that row material is available

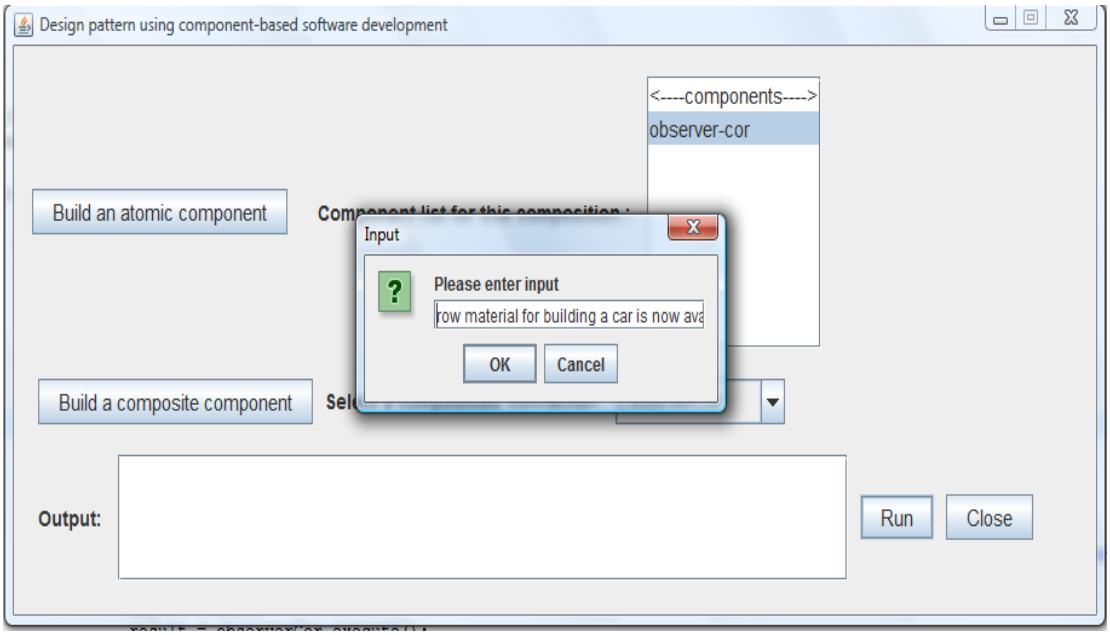

Figure 67:

- Two subscribers have reacted.

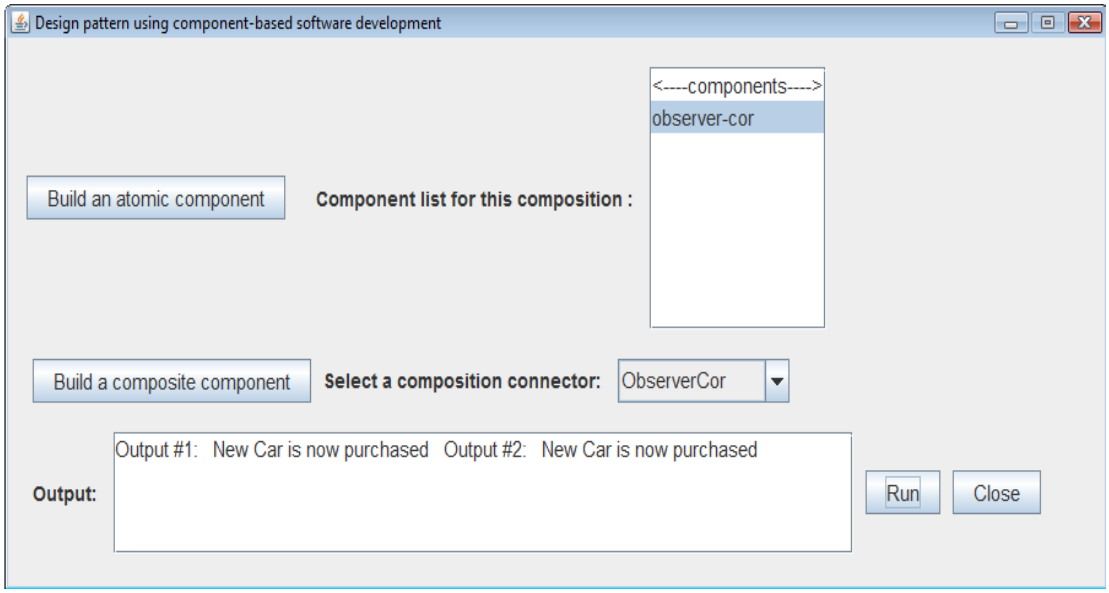

Figure 68:

10.4. FINAL SAY

In a nutshell, we started by going through some of the concepts of component-based software developments, followed by the notion of design patterns in object oriented paradigm. Next, I explain what the problem is of using design patterns in such an environment. Then, to remedy this I offer the solution of using pattern in the environment of component-based software development as a concrete reusable element, that is, a composition operator.

To put the issue into perspective, such approach towards patterns makes pattern reusable so that they are not required to be coded into each application. They can be designed, coded, and deposited into repository once, and then be retrieved indefinite number of times. Additionally, this approach enables pattern to be composed with other components like what we did in case of composing ObseverCC and CoR.

11. BIBLIOGRAPHY

1. *Exogenous Connectors for Software Components.* **Lau, K.-K, Velasco Elizando, P and Wang, Z.** s.l. : Springer-Verlag, 2005.

2. **Kaur, Kuljit, et al.** *Towards a Suitable and Systematic Approach for Component Based Software Development.* 2007 : World Academy of Science, Engineering and Technology.

3. *Component Technology.* **Steel, J.** London : International Data, 1996.

4. **Microsoft.** *The Microsoft Object Technology Strategy: Component Software.* 1996.

5. **Kruchten, Philippe.** *The Rational Unified Process An Introduction.* s.l. : Addison Wesley, 3rd. Eddition.

6. **Bachmann, Felix, et al.** *Technical Concepts of Component-Based Software Engineering.* s.l. : Carnegie Mellon University, 2000.

7. *Software Component Models.* **Lau, K.-K and Wang, Zheng.** s.l. : IEEE, 2007.

8. **Gamma, Erich, et al.** *Design Patterns: Elements of Reusable Object-oriented Software.* Holland : Addison Wesley, 1994.

9. Design Pattern. *wikipedia.* [Online] [Cited: 02 August 2009.] http://en.wikipedia.org/wiki/Design_pattern.

10. Design pattern (computer science). *wikipedia.* [Online] [Cited: 02 August 2009.] http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29.

11. **Bennett, Simon, McRobb, Steve and Farmer, Ray.** *Object-oriented System Analysis and Design.* s.l. : McGrawHill, 2006.

12. Architectural pattern (computer science). *wikipedia.* [Online] [Cited: 02 August 2009.] http://en.wikipedia.org/wiki/Architectural_pattern_%28computer_science%29.

13. *Formalising Design Patterns.* **Mikkonen, Tommi.** Tampere, Finland : IEEE, 1998.

14. *Towards Composing Software Components in Both Design and Deployment Phases.* **Lau, K.-K, Ling, L and Velasco Elizondo, P.** s.l. : Springer, 2007.

15. *A compositional approach to active and passive components.* **Lau, K.-K and Ntalamagkas, I.** s.l. : IEEE, 2008.

16. *Composing Components in Design Phase using Exogenous Connectors.* **Lau, K.-K, Ling, L and Wang, Z.** s.l. : IEEE, 2006.

17. *Composite Connectors for Composing Software Components.* **Lau, K.-K, Ling, L and Velasco Elizondo, P.** s.l. : Springer-Verlag, 2007.

18. **Shalloway, Allan and R. Trott, James.** *Design Patterns Explained.* s.l. : Addison Wesley, 2002.

19. **Buschmann, Frank, et al.** *Pattern-oriented Software Architecture: A System of Patterns.* s.l. : Wiley, 1996.

20. *Composite Design Patterns.* **Riehle, Dirk.** Ubilab : ACM, 1997.