AN ARCHITECTURE-BASED APPROACH FOR EMBEDDED SYSTEMS

Layered Architecture

Tran Xuan Hoang
Email: hoangtx@jaist.ac.jp

Last compiled and converted to PDF: August 12th, 2017

1. Introduction

This research introduces layered software architecture and its application in constructing embedded software systems. We will construct a simulation software to simulate the operations of a factory robot in order to understand the advantages and the benefits of using layered architecture. Finally, a comparison between layered architecture and use case driven architecture will be shown through an additional update of the simulation software.

2. Layered Architecture

Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility. Communication between layers is explicit and loosely coupled. Layering our application appropriately helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.

The layered architectural style has been described as an *inverted pyramid of reuse* where each layer aggregates the responsibilities and abstractions of the layer directly beneath it. With strict layering, components in one layer can interact only with components in the same layer or with components from the layer directly below it. More relaxed layering allows components in a layer to interact with components in the same layer or with components in any lower layer.

The layers of an application may reside on the same physical computer (the same tier) or may be distributed over separate computers (*n*-tier), and the components in each layer communicate with components in other layers through well-defined interfaces. For example in the book *Microsoft Application Architecture Guide* [2], a typical Web application design consists of a presentation layer (functionality related to the UI), a business layer (business rules processing), and a data layer (functionality related to data access, often almost entirely implemented using high-level data access frameworks).

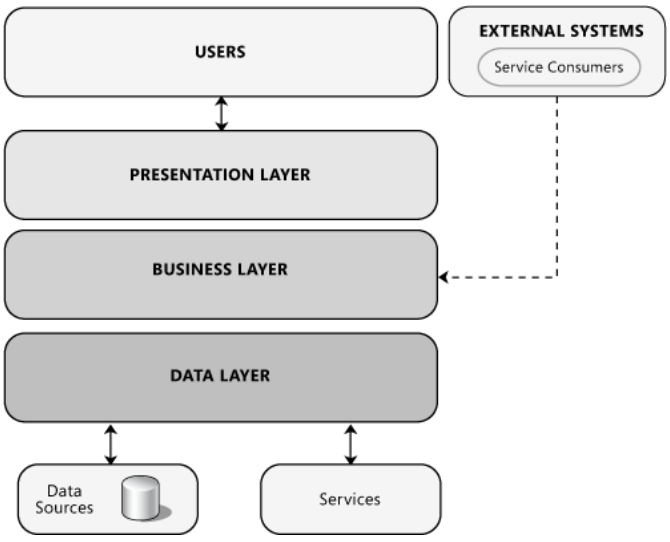


Figure 1: The logical architecture view of a layered system

Figure 1 (extracted from chapter 5 of [2]) depicts a simplified, high level representation of these layers and their relationships with users, other applications that call services implemented within the

application's business layer, data sources such as relational databases or Web services that provide access to data, and external or remote services that are consumed by the application.

There are many different ways to group related functionality into layers. However, separating an application into too few or too many layers can add unnecessary complexity; and can decrease the overall performance, maintainability, and flexibility. Determining the balance of layering appropriate for our application is a critical first step in determining our layering strategy.

Determine Rules for Interaction between Layers

When it comes to a layering strategy, we must define rules for how the layers will interact with each other. The main reasons for specifying interaction rules are to minimize dependencies and eliminate circular references. For example, if two layers each have a dependency on components in the other layer we have introduced a circular dependency. As a result, a common rule to follow is to allow only one way interaction between the layers using one of the following approaches:

- **Top-down interaction**. Higher level layers can interact with layers below, but a lower level layer should never interact with layers above. This rule will help us to avoid circular dependencies between layers. We can use events to make components in higher layers aware of changes in lower layers without introducing dependencies.
- **Strict interaction**. Each layer must interact with only the layer directly below. This rule will enforce strict separation of concerns where each layer knows only about the layer directly below. The benefit of this rule is that modifications to the interface of the layer will only affect the layer directly above. Consider using this approach if we are designing an application that will be modified over time to introduce new functionality and we want to minimize the impact of those changes, or we are designing an application that may be distributed across different physical tiers.
- **Loose interaction**. Higher level layers can bypass layers to interact with lower level layers directly. This can improve performance, but will also increase dependencies. In other words, modification to a lower level layer can affect multiple layers above. Consider using this approach if we are designing an application that we know will not be distributed across physical tiers (for example, a stand-alone rich client application), or we are designing a small application where changes that affect multiple layers can be managed with minimal effort.

3. Factory Automation Software Product Line

In chapter 15 of the book *Designing Software Product Line with UML From Use Cases to Pattern-Based Software Architectures* [3], Hassan Gomaa introduces the design model for factory automation software product line. Layered component-based architecture is used to model the software system, in which 4 types of components and 6 layers are determined.

4 types of components are as follows:

- *Server components:* Create and save information about workflow plans or products.
- *User interface components:* Help users interact with the software system via GUI.
- *Control components:* Control the relevant business logic of the software system.
- *Coordinator components:* Coordinate the moves or operations of the factory robots.

These components are structured into the layered architecture such that each component is in a layer where it needs the service provided by components in the layers below but not the layers above. The rule for interaction between layers in this architecture is a hybrid between *top-down interaction* and *loose interaction* described in section 2.

The following figure shows 6 layers and the rule for interaction between them:

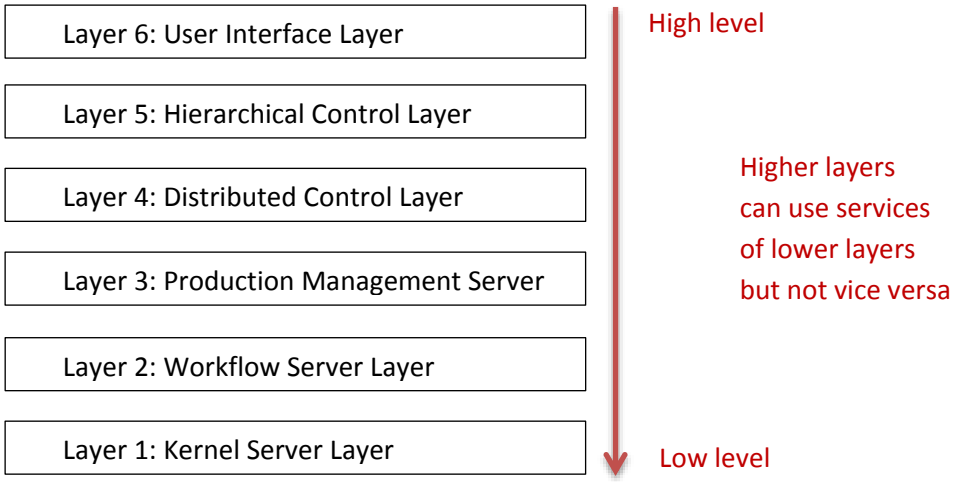


Figure 2: Layered architecture and the rule for interaction between its layers

Layer 1 – Kernel Server Layer: plays the role of connecting between the software systems with the hardware of the factory robots. This layer contains coordinator components that can be seen as APIs for the software system to interact with the factory robot.

Layer 2 – Workflow Server Layer: creates and saves workflow plans for the factory robots in order to make concrete products. Normally, this layer consists of server components and it can uses services of layer 1 to generate the workflow plans (the instructions that the factory robots will follow to produce products).

Layer 3 – Production Management Server: plays the role of holding information about products that the factory robots will produce.

Layer 4 – Distributed Control Layer: provides distributed control for distributed software system. When the factory robots need a complex and distributed software system, this layer will provide a consistent way for managing all the logical control businesses between the robots so that these robots will smoothly co-operate together.

Layer 5 – Hierarchical Control Layer: consists of control components in order to provide hierarchical and logical control business inside each robot so that each robot will know exactly what it has to do, and how it will do to complete its tasks.

Layer 6 – User Interface Layer: provides graphical user interface that helps managers/engineers/ workers of the company interact with the software system graphically. This layer consists of user interface components and it allows managers/engineers/workers to enter inputs as well as demonstrates all the process of robots' operations.

4. Case Study: Simulation Software of Factory Robot Hand

In this section, we will design and construct a simulation software for a factory robot using layered component-based architecture described in section 3. The simulation software will be designed by applying the rules of layer separation (6 layers as described in section 3) and interaction between layers (hybrid of top-down interaction and loose interaction as described in section 2).

The automation software that we will design is for a factory robot that picks and moves some items which have the same color from a given place to the destination place. The software will be designed and implemented by using layered architecture. In the layered architecture, we divide the software into components, including: server components, user interface components, control components and coordinator components. The components, then, are structured into the layered architecture such that each component is in a layer where it needs the services provided by components in the lower layers but not the higher layers. Since our software is for only a robot, and users can enter the information of items (products) from GUI, our software will consist of four layers, including: layer 1 kernel server layer, layer 2 workflow server layer, layer 5 hierarchical control layer, and layer 6 user interface layer.

Layer 1: Kernel Server Layer

Description: This layer contains coordinator components that represent operations the robot's hand can perform. In addition, layer 1 also contains server components so that higher layers can use services of these components to save the status of the table and color items on the table.

Components: MoveTop, MoveBottom, MoveLeft, MoveRight, Open, Close, Up, Down, TableCell, Items, RobotHand.

Require Services or Data (Input): None

Providing Services (Output): This layer provides all services of moving, picking up, putting down of the robot's hand. It also provides services for representing and saving information relating to the status of the table and items and information about robot's hand.

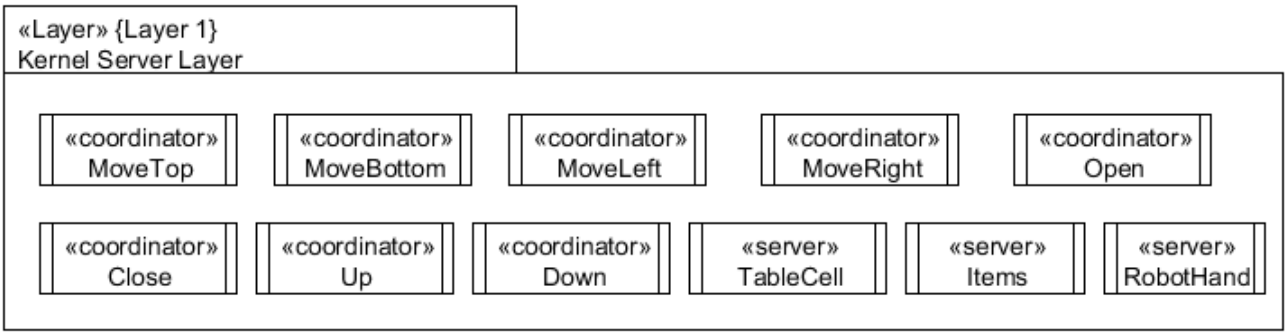


Figure 3: Coordinator components and server components of layer 1

These components can be used by all upper layers. Coordinator components are used to order the robot hand move, pick up, put-down items. Server components are used to represent and save information about the table, the items on the table, and the robot's hand (such as: position, item on its hand, etc...).

Layer 2: Workflow Server Layer

Description: Creating and saving plan of flow of operations that robot's arm will perform in order to move all necessary items to the desired positions.

Components: OperationFlowPlanning

Require Services or Data (Input): This layer requires the MoveTop, MoveBottom, MoveLeft, MoveRight, Open, Close, Up, Down, TableCell, Items and RobotHand components from layer 1.

Providing Services (Output): This layer provides service to make the flow of operations of the robot's hand. The other top layers will use this service as the instruction to move all necessary items.

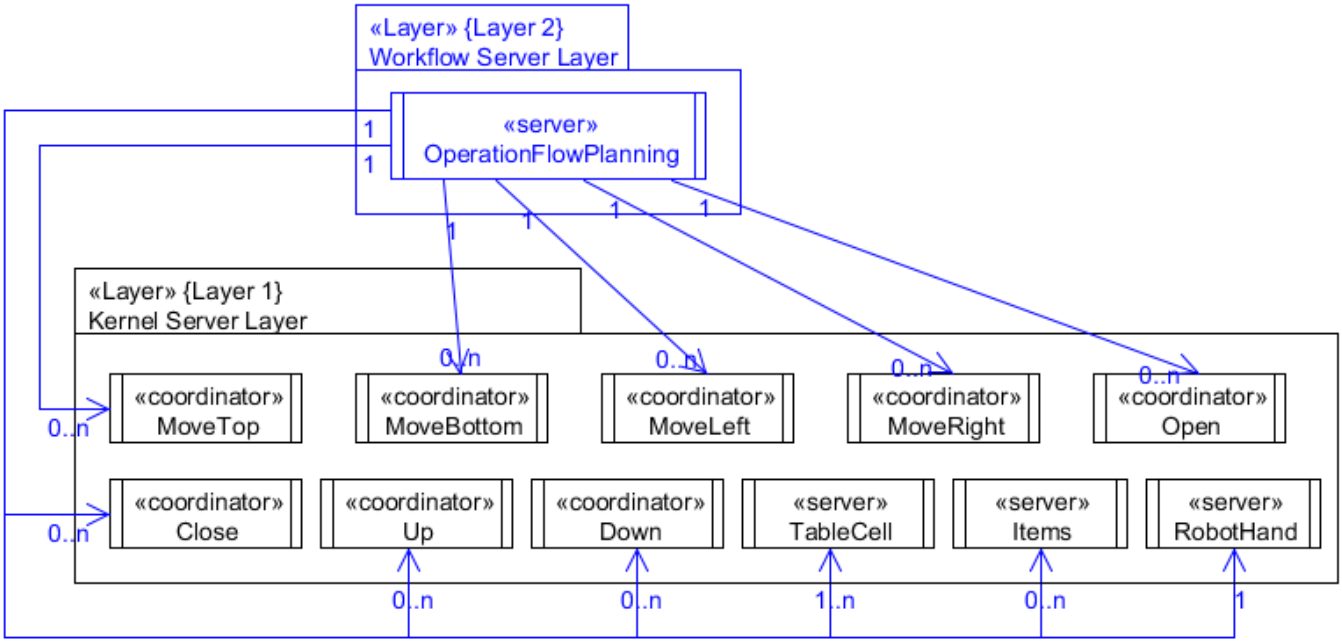


Figure 4: Layer 2 and its dependencies with layer 1

Component OperationFlowPlanning will need all the coordinator components of layer 1 to make a plan of moving the robot's hand. Furthemore, OperationFlowPlanning will also need server components of layer 1 in the calculation of necessary operations for the robot.

Layer 3: Production Management Server

Description: Normally, this layer consists of components that represent information about items (location, quantity, type, etc...) at the beginning when the robot's hand has not performed any operations. In our simulation software, the information is entered via GUI by users. Therefore, this layer does not play any roles in our simulation software.

Layer 4: Distributed Control Layer

Description: Normally, this layer consists of components that that provide distributed control for the distributed software systems. In our factory robot automation software, this layer does not play any roles since our software is not designed for distributed system.

Layer 5: Hierarchical Control Layer

Description: This layer consists of components that control the process of moving items on the table to the desired locations and the control of simulating this process.

Components: ControlRobotArm, RBHSimulation, RBHOperationsSimulation

Require Services or Data (Input): This layer requires all components in layer 1 except Items component. It also needs the OperationFlowPlanning component in layer 2 to obtain the flow of operations that will be executed.

Providing Services (Output): This layer provide the control for logical business inside the software system, including the control for moving the robot's hand, picking up and putting down items; and the *control for simulating all operations* of the robot's hand graphically.

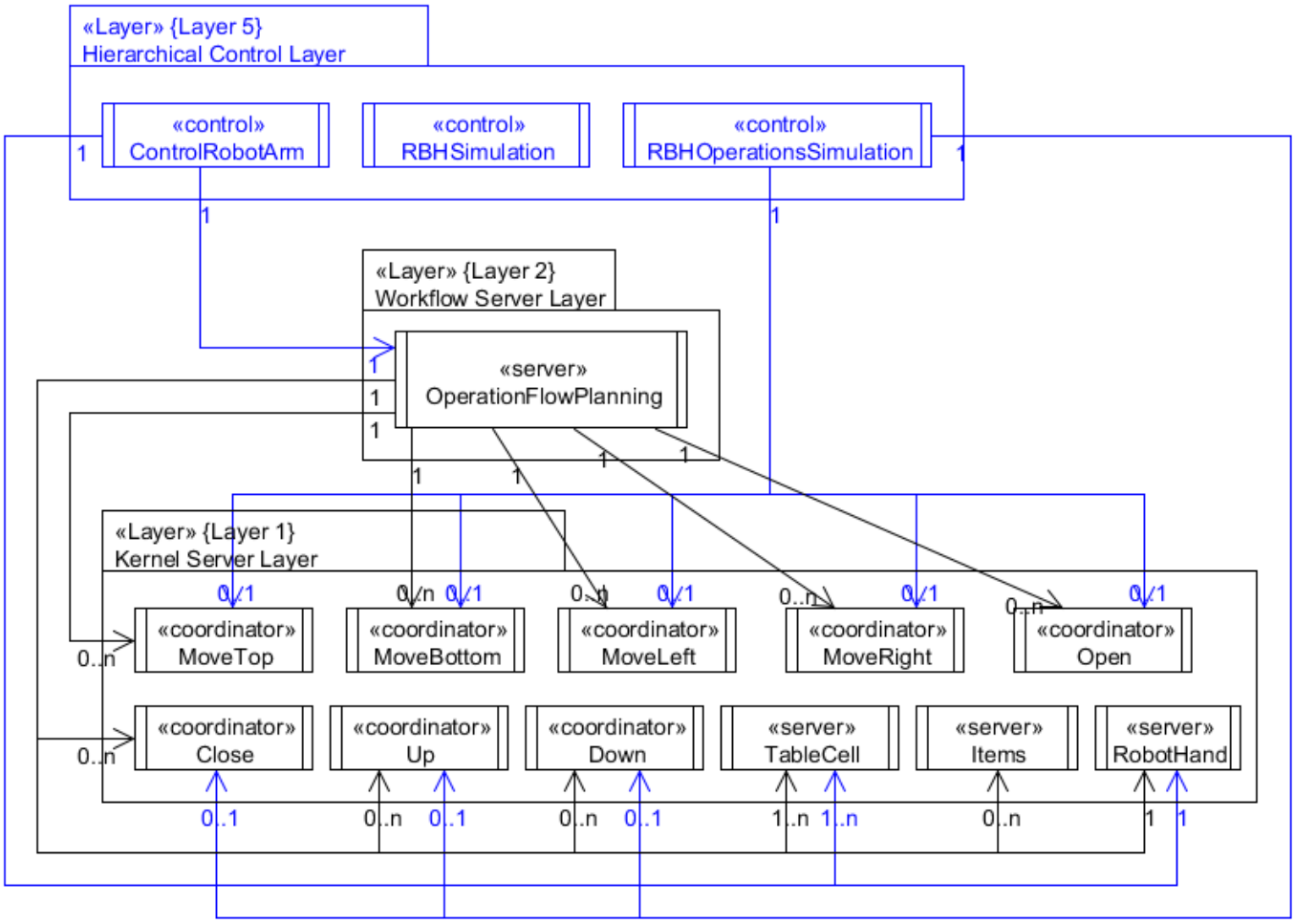


Figure 5: Layer 5 and its dependencies with layer 2 and layer 1

Component ControlRobotArm is the main control inside the software system. It takes the flow of operations needed to move all items to desired positions and coordinates both the robot's hand and the simulation of these robot's operations. Component RBHOperationsSimulation receives control signals from ControlRobotArm and creates the corresponding simulations of the robot's operations. Component RBHSimulation actually is a supporter component that contains information to simulate the robot's hand and is used by ControlRobotArm and RBHOperationsSimulation components.

Layer 6: User Interface Layer

Description: This layer consists of user interface components that allow users to enter parameters into the robot control system and display the status of the table with items while picking up and moving items are occurring.

Components: ViewTable

Require Services or Data (Input): This layer requires the ControlRobotArm component in layer 5, the TableCell, Items and RobotHand components in layer 1.

Providing Services (Output): This layer provides all necessary graphical user interfaces to display the process of picking up and moving items as well as the states of the table.

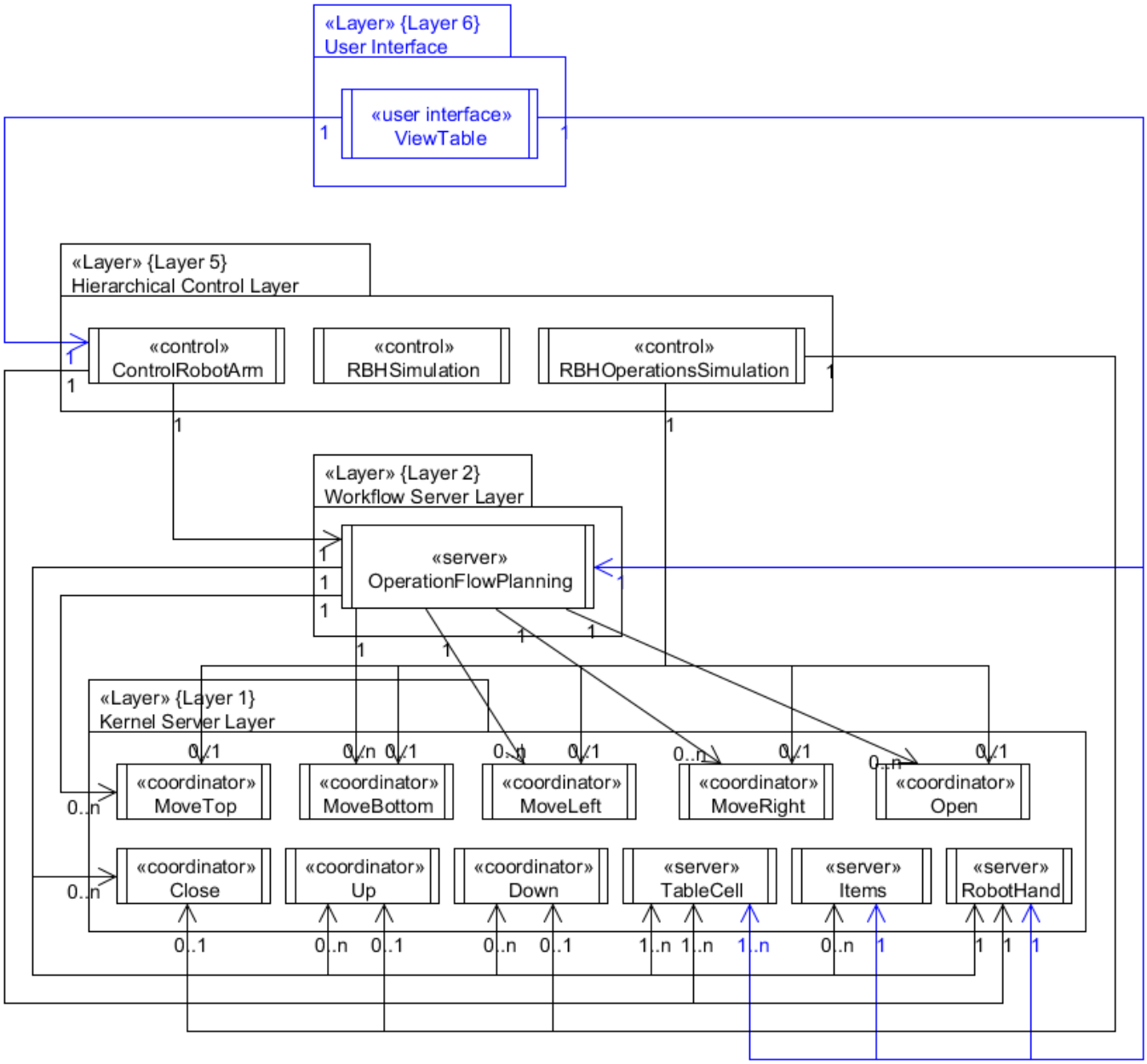


Figure 6: Layer 6 and its dependencies with lower layers

Layered Architecture of the factory robot automation software

The following layered architecture diagram shows the dependencies between layers in the factory robot automation software. This diagram is a summary of all the previous diagrams.

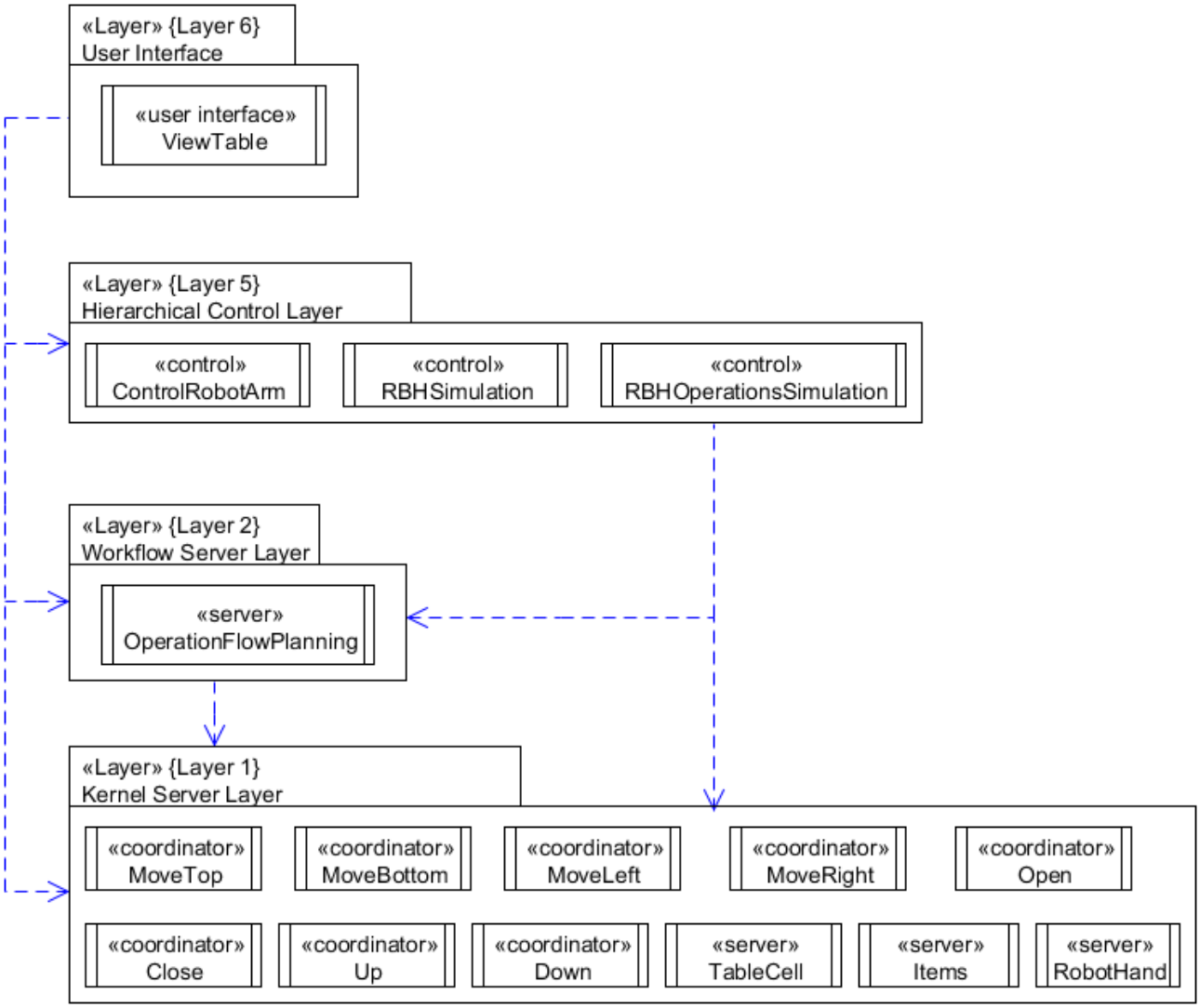


Figure 7: The overall layered architecture of the simulation software for the factory robot
The blue dashed lines represent the dependencies between layers: Higher layers can utilize services of components of lower layers but not vice versa.

Implementation of the simulation software

All the designs described in this section were implemented in Java programming language, in which each layer is represented as a *package*, each component is represented as a *class*, and each service is represented as a *method* in Java. Layer 6, layer 5, layer 2, layer 1 correspond to package layer6, layer5, layer2, layer1 respectively in the Java source code. Classes of higher packages can import classes of lower packages but not vice versa, and all the importations in each class source file follow the design in the previous figures. For example, class ControlRobotArm in package layer5 will import classes RobotHand, TableCell of package layer1, and class OperationFlowPlanning of package layer2.

5. Results and Discussion

In this section, we introduce Stucture101 tool [4] which is used to analyze and evaluating the architecture of our simulation software developed in the previous section. Structure101 version for Java is a tool that lets the software development team builds the model of architecture of the software system direct from the source code. It can model the source code in a modular hierarchy and measure some metrics relating to characteristic of the architecture of the software.

The input of Structure101 is the Java byte code, so we will use all compiled classes developed in the previous section. Structure101 returns us the following model of our software's architecture:

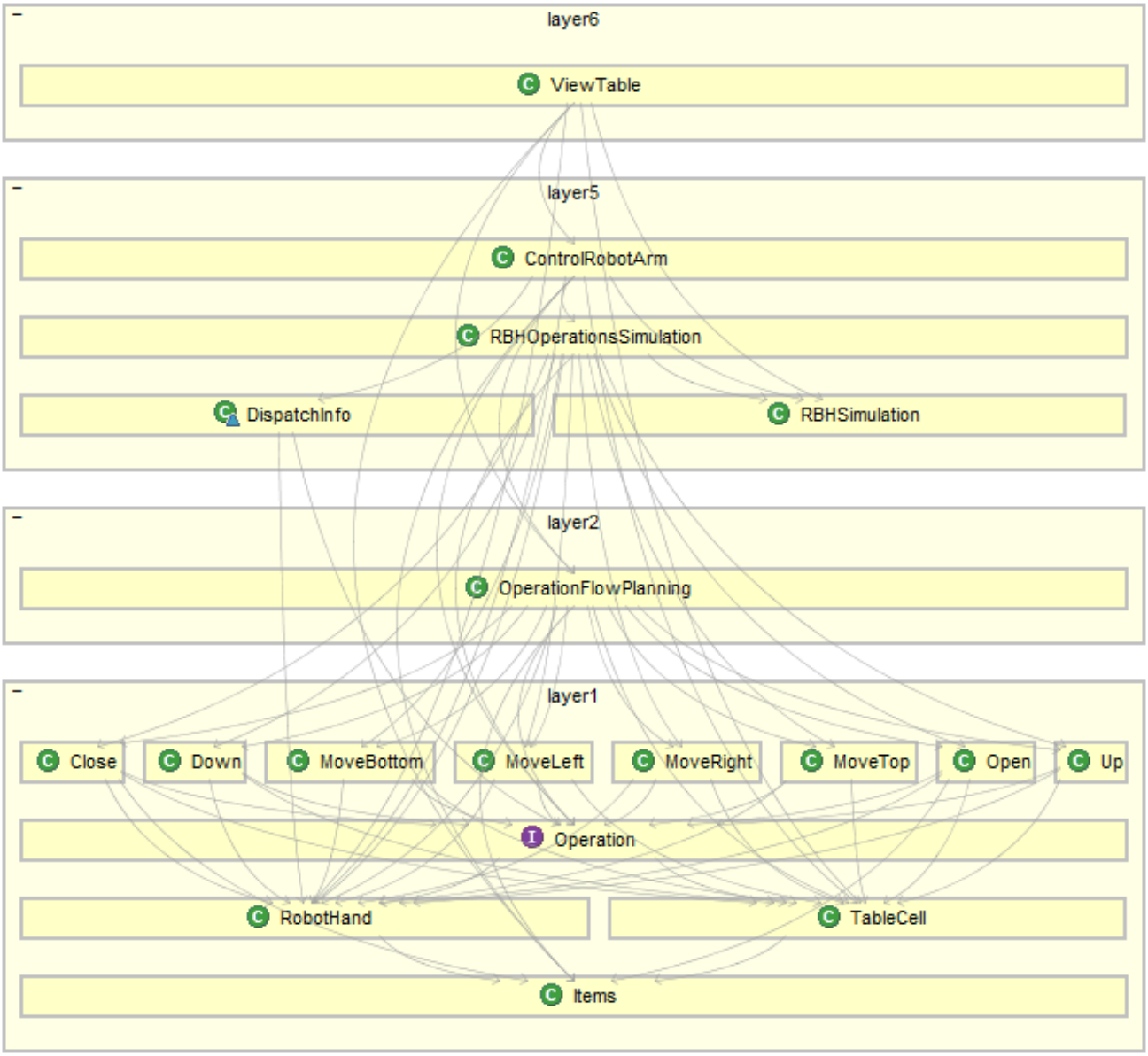


Figure 8: The architecture of the simulation software returned by Structure101 tool

The architecture returned by Structure101 tool and the architecture we designed in figure 7 are the same: 4 layers; every component of the architecture in figure 7 appears as a class in the architecture in figure 8, except interface Operation and private class DispatchInfo are added because they make the

implementation in Java more appropriate; all the links (gray arrowed lines) point from higher layers to lower layers that mean higher layers' classes use some methods/objects/enums of lower layers – these links play the same roles as dependencies between layers in figure 4, 5, 6.

The following figure shows the table of dependencies between each class with other classes. The number in each cell represents the weight of the dependency between class at the corresponding column and the corresponding row (class at column will use services of class at row). The bigger weight the cell has, the stronger dependency between classes is.

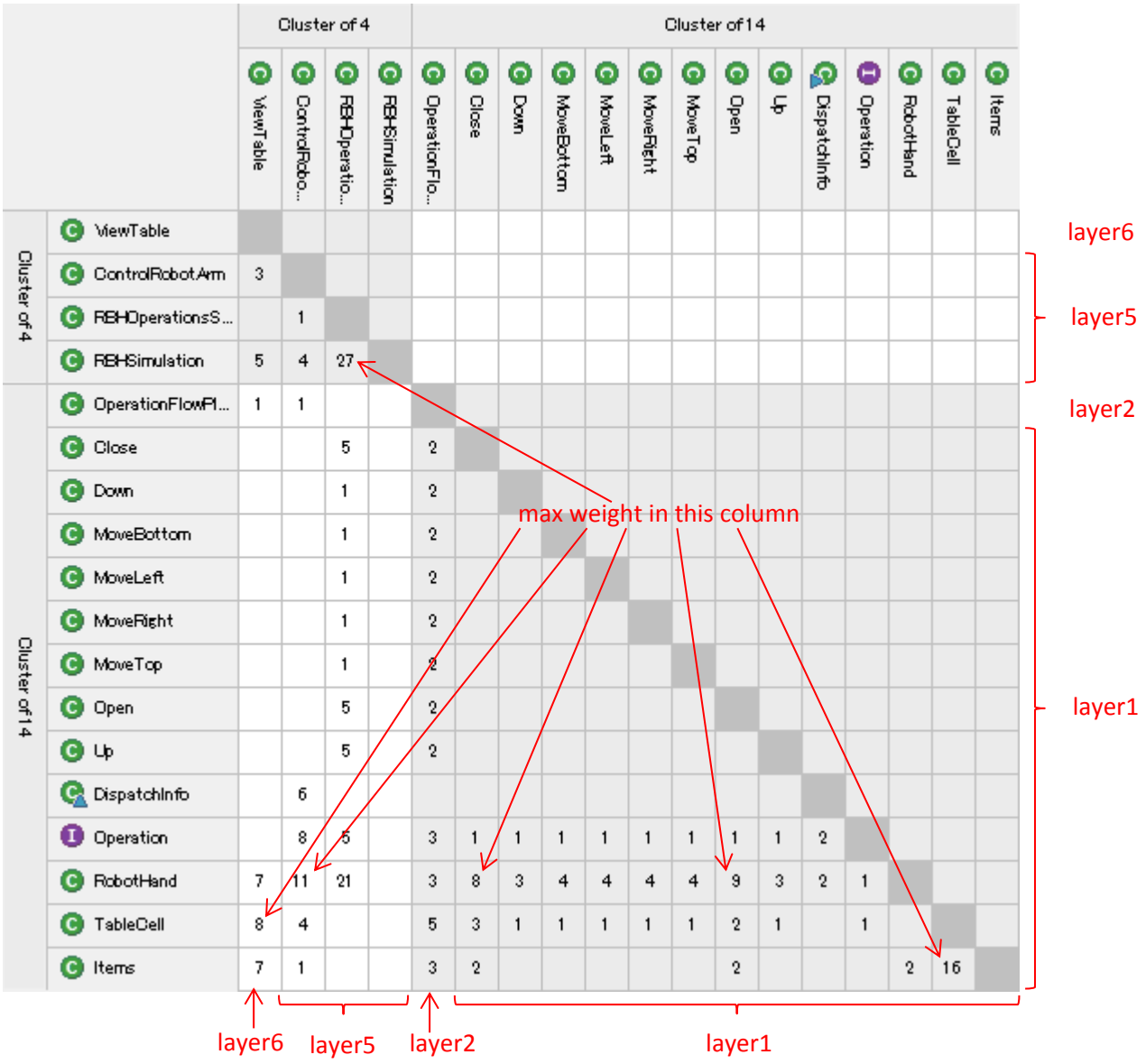


Figure 9: Dependencies between classes represented as weight numbers

From the above table we can easily see that: the max weights almost appear in layer1, in other words layer1's components are used by all other layers so it plays an important role in building the foundation of our software. Every other layer has a strong dependency with layer1. Another important result is that the biggest weights of dependencies of two classes appear for the two classes in the same layer – this means the separation of modules/classes is fairly appropriate. Finally, the Structure101 tool automatically

combines layer1 and layer2 into a group, layer5 and layer6 into a group – this is also appropriate for our software since layer6 user interface and layer5 hierarchical control are the two highest layers while layer2 workflow server and layer1 kernel server are the two lowest layers in our architecture model.

The final remarkable result returned by Structure101 tool is the quality of the source code of our software, including the percent of tangled and fat source code packages and class files. Figure 10 shows that our source code is well organized and there are no tangled source code (here, a "tangle" is a set of items, packages/classes/methods, that forms a cyclic dependency), and there are no fat source code (here, "fat" is too much complexity in one place for a developer to easily understand – when too much code accumulates in one method, we instinctively extract out a cogent sequence of instructions into a new function with clearly identified parameters. The 2 functions working together are easier to understand than the original more complex one).

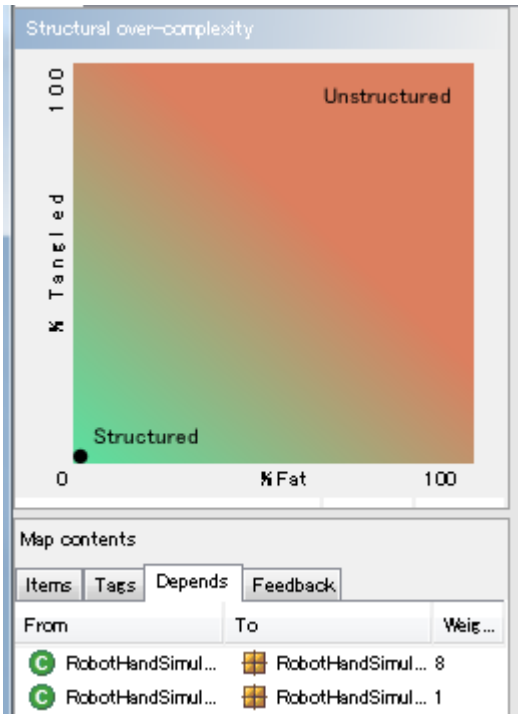


Figure 10: The structural over-complexity of the source code
The black point indicates that our source code is well organized with no tangled and fat source code.

6. Comparison between layered architecture and use case driven architecture

In this section, we re-design our software for the factory robot by using use case driven architecture proposed by Jacobson [1]. Use case driven approach was proposed at the very beginning days of object oriented technology and have been studied for a long time. It can be seen as a traditional architecture in object oriented technology. After we finish the design by using the use case driven architecture, we will show out some cases that make this traditional design is less flexible and less modifiable than the layered architecture.

The class diagram for the simulation software of the factory robot designed by using use case driven architecture is showed in figure 11. The table, robot hand, items, operations are represented as classes Table, RobotHand, Item, interface Operation and concrete classes Open, Close, MoveLeft, MoveRight, MoveTop, MoveBottom, Up, Down. To simulate the table, the robot hand, items, we created classes TableSim, RBHSim, ItemsSim. To simulate operations of the robot we add method simulate(ts: TableSim, rbhs: RBHSim) to interface Operation and each subclass of Operation will implement this method.

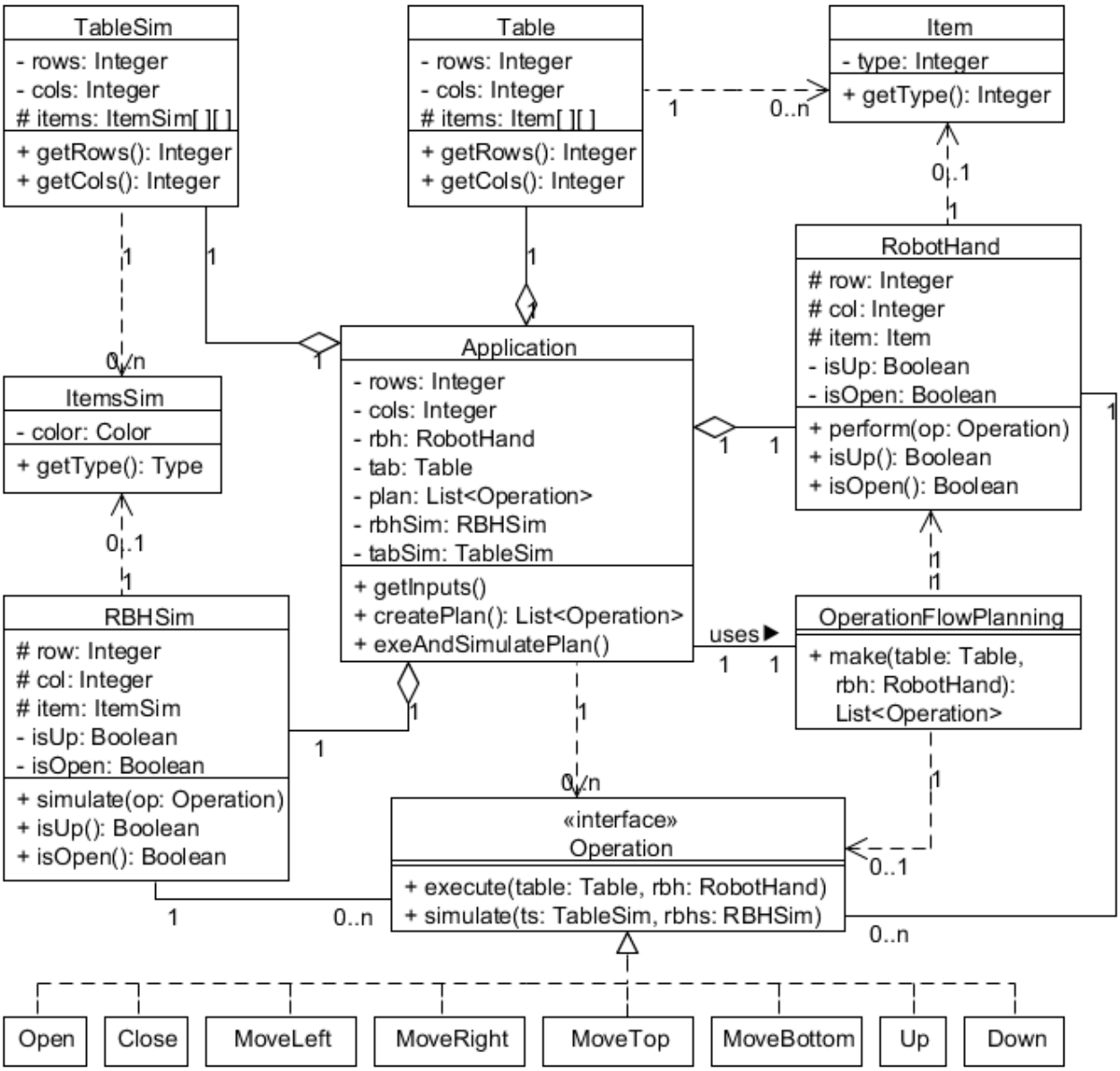


Figure 11: Class diagram designed by using use case driven architecture

RBHSim, TableSim, ItemSim are classes that play the role of simulating RobotHand, Table, Item graphically. Open, Close, MoveLeft, MoveRight, MoveTop, MoveBottom, Up, Down are concrete classes that implement interface Operation, so each of the class has execute() and simulate() methods.

Suppose that, now we need to update the graphical user interface of our software. We also change the simulation interface of the table, the robot hand and items so that they will become more appropriate and compatible with the new GUI. Method exeAndSimulatePlan() in class Application will be the starting point of both the execution of plan of operations and the simulation of these operations until the robot moves all items to desired positions. Figure 12 in the next page is the sequence diagram that shows the method calls while exeAndSimulatePlan() is beeing executed.

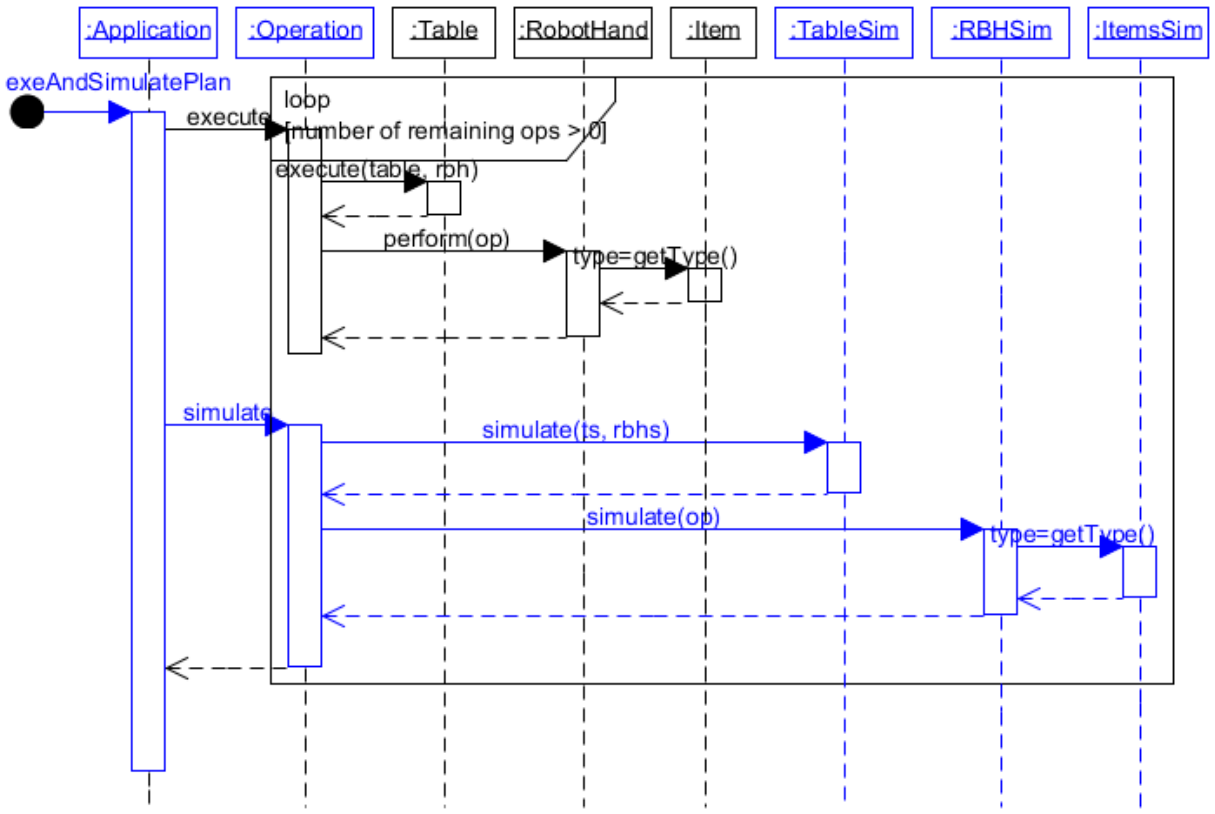


Figure 12: Sequence diagram for executing and simulating the flow of operations
The blue part contains classes that are affected if the GUI and the simulation of robot are changed

The largest advantage of layered architecture is that, the modification, update or addition of functions of the software will affect only a small number of components. From the sequence diagram in figure 12, if the GUI and the simulation of the table, the robot hand and items are changed, then classes Application, Operation, TableSim, RBHSim and ItemSim need to be modified accordingly. In addition, because classes Open, Close, MoveLeft, MoveRight, MoveTop, MoveBottom, Up, Down implement interface Operation, these classes also need to be modified (namely, method simulate(ts: TableSim, rbhs: RBHSim) need to be modified). On the other hand, if we use *layered architecture design* described in the previous sections, then all we need to do is modifying class ViewTable of layer6 since layer6 plays the role of all tasks related to GUI as well as simulation of the table, the robot hand and items. Figure 13 in the next page depicts the necessary modifications of classes in the use case driven architecture and layered architecture. We can easily see that, the modification of only one class ViewTable in the layered architecture model is much easier than the modification of 13 classes (Application, TableSim, RBHSim, ItemSim, Operation, Open, Close, MoveLeft, MoveRight, MoveTop, MoveBottom, Up, Down) in the use case driven architecture model. The advantage of layered architecture in this case is really useful. This demonstrates that the layered architecture is more applicable and appropriate than the use case driven architecture in design and implementation of software product line for the company robot systems.

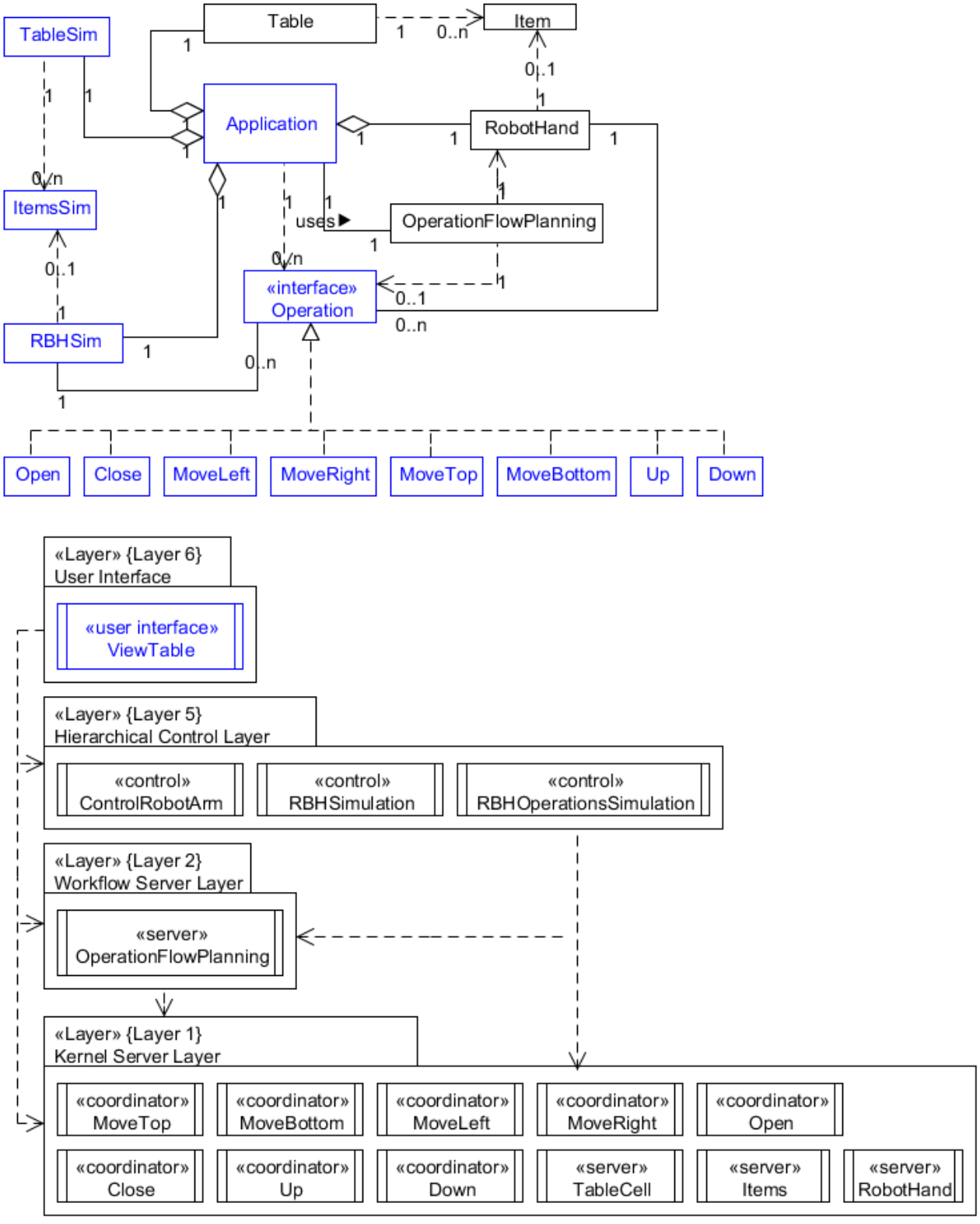


Figure 13: Impact analysis of classes between use case driven architecture and layered architecture. Blue-color classes in each architecture model need to be modified. This figure shows that, for use case driven architecture we need to modify 13 classes while for layered architecture we need to modify only 1 class.

7. Conclusions

In this research, we have focused on layered architecture in embedded systems, namely simulation software systems for the factory robot. Through this research, we have studied some important aspects of layered architecture. Layers help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sub layers, with each sub layer performing a specific type of task. The main benefits of using layered architecture are:

- Layers allow changes to be made at the abstract level. We can increase or decrease the level of abstraction we use in each layer.
- Allows us to isolate technology upgrades to individual layers in order to reduce risk and minimize impact on the overall system.
- Separation of core concerns helps to identify dependencies, and organizes the code into more manageable sections.
- Promote reusability. For example higher layers can reuse services of lower layers.
- Increased testability arises from having well-defined layer interfaces, as well as the ability to switch between different implementations of the layer interfaces.

We have also pointed out the advantages that make layered architecture become more flexible, modifiable and appropriate than the traditional use case driven architecture in the design and implementation of the software product line for the factory robot systems.

8. References

[1] Jacobson, Ivar. *Object Oriented Software Engineering: A Use Case Driven Approach*. 1st ed. Addison-Wesley Professional, 1992. 109-333. Print.

[2] Microsoft Patterns & Practices Team. "Layered Application Guidelines." *Microsoft Application Architecture Guide*. 2nd ed. Microsoft Press, 2009. 55-66. Print.

[3] Gomaa, Hassan. "Factory Automation Software Product Line Case Study." *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. 1st ed. Addison Wesley, 2004. 505-612. Print.

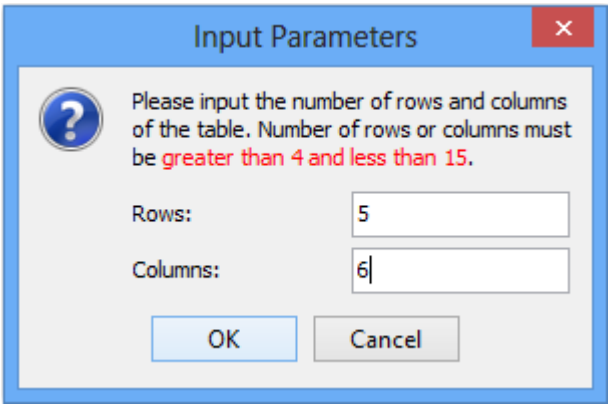
[4] *Structure101 Software Architecture Development Environment*. Web. https://structure101.com/

Appendix - Simulation Software

Step 1: Input the number of rows and columns of the table

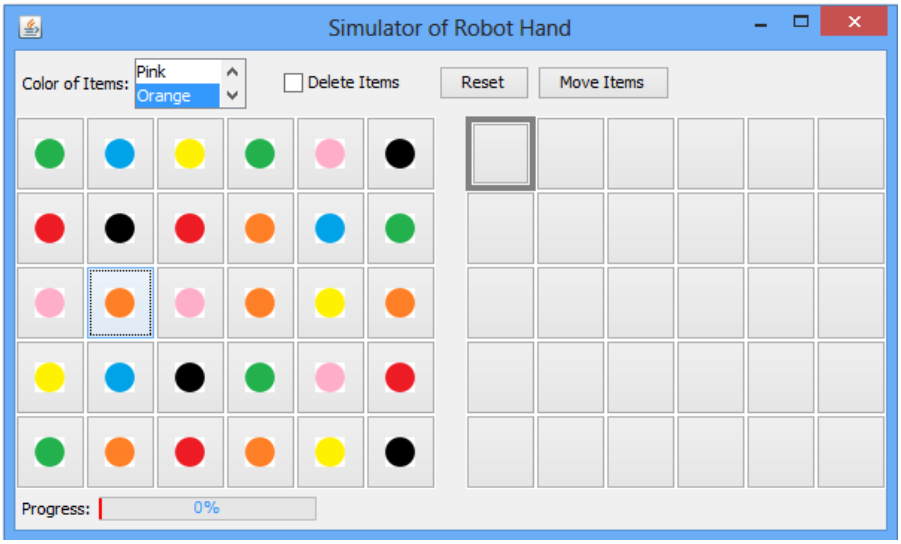
There are two ways to start the simulation software:

- Click on file run.bat in folder `~/bin`
- Change directory in command prompt or terminal to `~/bin` directory then type `java layer6.ViewTable`



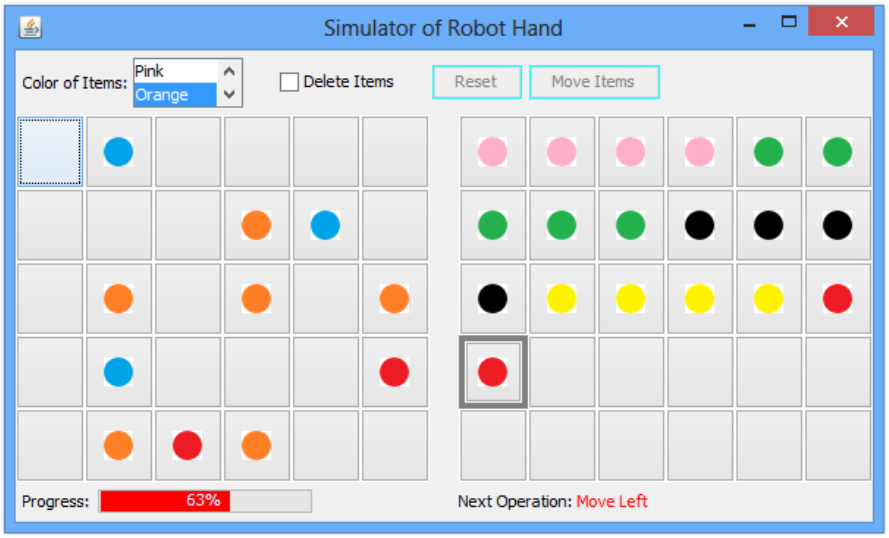
Step 2: Enter all items that robot needs to move

Type of items are represented as different colors, place of each item is represented as coordinates on the left table. Select one color from the drop down list of Color of Items, then click on cell that the item belongs to. If an item need to be deleted, click Delete Items check box then click the cell of the item that need to be deleted. Click Reset button to clear all items and reset the robot.



Step 3: The robot will start moving all items to desired positions

By clicking Move Items button, we start the robot's operations execution. The robot hand will find and move all items that have the same color to the consecutive positions on the right-hand-side table. The Progress bar will show the percent of work that the robot has completed, and the label Next Operation will show the next operation that the robot hand will perform.



Step 4: The robot finishes moving items

When the robot finishes performing all operations of its work flow plan, all items will be moved and sorted as their types in the right-hand-side table. Click Reset button to come back to Step 2 if we want to start with moving and sorting other items.