

Faculty of Information and Communication Technologies
Centre for Information Technology Research

Research Program in Component Software and Enterprise Systems (CeCSES)

CeCSES Report: SUT.CeCSES-TR009

The Implementation of Message Synchronisation, Queuing and Allocation in the ROAD Framework

Linh Duy Pham, Swinburne University of Technology
Alan Colman, Swinburne University of Technology
Jun Han, Swinburne University of Technology

Version 1.0

02 April 2006



SWIN
BUR
NE

SWINBURNE UNIVERSITY
OF TECHNOLOGY

Abstract

In a dynamic environment where requirements change and components' performance varies, the system should be able to configure its internal structure in order to maintain the overall satisfactory output. An approach to this adaptability has been introduced in the ROAD framework. Our prototype of the ROAD framework was implemented using a number of techniques, including Association-Aspects for implementing ROAD contracts. The performance monitoring and adaptability was realised using message interception for various synchronisation modes and message allocation/routing mechanisms inside the ROAD framework. The prototype demonstrates the ROAD framework's ability to monitor the system performance at runtime, and ability to react accordingly by changing players, creating new roles and contracts in order to utilise the capability of components and achieve/maintain system goals. The current stage of the implementation together with various alternatives and limitations are presented and discussed in this report. The implementation has proved the feasibility of the conceptual design of the ROAD framework and also opened up a number of areas that need to be investigated in the future.

Table of Contents

1	Introduction	1
2	Summary of ROAD Framework	2
2.1	Roles	2
2.2	Players	2
2.3	Contracts	2
2.4	Control and Communication Acts (CCAs)	3
2.5	Composites	3
2.6	Adaptive Behaviour	3
3	Scope	5
3.1	Initial Scope	5
3.2	Revised Scope	5
4	Issues and Solutions	6
4.1	Message Routing	6
4.2	Work Allocation	7
4.3	Message Queues in Roles	7
4.4	Synchronisation Types and Performance Measurement	8
4.5	Pull vs. Push Models	9
5	Results	11
5.1	State of ROAD Framework	11
5.2	ROAD Domain Specific Application	12
5.2.1	Widget Making Department	13
5.2.2	UML Class Diagram of Widget Making Department	14
5.2.3	Test Harness	14
5.2.4	Test Harness Output	15
5.2.5	Conclusion	18
6	Future Work	19
	References	19
	Appendices	20
	Appendix A – Test Harness Code	20
	Appendix B – Test Harness Output	23

List of Figures

FIGURE 1 MESSAGE ROUTING IN A ROAD STRUCTURE	6
FIGURE 2: DIFFERENT SYNCHRONISATION TYPES (FROM [1])	8
FIGURE 3: ROAD FRAMEWORK SIMPLIFIED UML DIAGRAM	12
FIGURE 4: WIDGET MAKING DEPARTMENT (FROM [1])	13
FIGURE 5: WIDGET MAKING DEPARTMENT UML SIMPLIFIED CLASS DIAGRAM	14
FIGURE 6: INITIAL COMPONENTS INSIDE WIDGETDEPCOMPOSITE INSTANCE	15
FIGURE 7: THE CONFIGURATION OF WIDGETDEPCOMPOSITE IS CHANGED DYNAMICALLY	16
FIGURE 8: ANOTHER CONFIGURATION OF WIDGETDEPCOMPOSITE INSTANCE AT RUN TIME	17

The Implementation of Message Synchronisation, Queuing and Allocation in the ROAD Framework

1 Introduction

Component-based software engineering (CBSE) presents an efficient way to reuse thus improve productivity and save software development costs and efforts. However, much work has been devoted to solve the component compatibility issues in both functional and non-functional aspects. ROAD (Role Oriented Adaptive Design) framework developed by Colman and Han [1] presents a solution to this problem by manipulating and coordinating components based on the roles that they play in the system. The framework monitors and regulates non-functional properties by means of contracts between different roles in the system. In the case of changing requirements and/or a component fails to maintain a satisfactory level of performance, ROAD will replace it with a better performance component (if such component exists and is available) or ROAD will restructure the role relationships in order to utilise the available components.

The purpose of this report is to document the results of a semester long project in exploring various mechanisms for extending the implementation the ROAD framework. This project was part of the assessment requirements of the subject HIT 4071 Research Project. The conceptual level design of the ROAD framework is presented in Colman and Han [1], and an understanding of these concepts is assumed in this report. The implementation of the ROAD framework was done by using AspectJ [2] and its extension - Association Aspects [3]. More details of this implementation can be found in [4]. This report describes the work done to extend the implementation of the framework to include various schemes for message handling.

The development process did not follow any strict Software Development Life Cycle, however it can be described as an informal Iterative Model. In this process, various design solutions were proposed to messaging handling issues raised by the ROAD approach. These candidate solutions were assessed, implemented and evaluated. The research challenges included the following: correctly modelling the concurrency property or synchronisation types (i.e. one-way, synchronous, asynchronous, or deferred-synchronous messaging); correctly monitoring the performance of components in different synchronisation types and safely replacing the under-performing components. The approach was to use message interception and reflection to solve the above problems.

2 Summary of ROAD Framework

As software systems become more open, distributed, pervasive, and connected, they need to be able to adapt to their dynamic environments. One approach to build such an adaptive system is to define the system in terms of roles and their relationships. Such definition will result in loosely-coupled components. Existing or newly available components can be dynamically chosen to perform these roles. By restructuring the relationships between roles, the association between components can be changed dynamically.

The ROAD framework follows this approach. It extends the work on object-oriented role and associative modelling. In this framework, the elements that are being coordinated are *roles* played by *players* (objects, components, agents, or humans). Below is a summary of the main components in the framework. For a complete and more detail description, refer to [1,5,6].

2.1 Roles

The ROAD design is based on an organisational perspective of a system, i.e. it views a software system as an *organisation* consisting of inter-related *roles*. Roles are first-class runtime entities that can be played by various players at different times. In general, *organizational* role defines an abstract function or a 'position' within an organization, while role-players "do the work".

The ROAD approach distinguishes two types of organisational roles, namely *functional*-roles and *organiser*-roles. *Functional roles* (or more properly *domain* function roles) are focused on first-order goals, that is, on achieving the desired application-domain output. Functional roles are associated to one another by *contracts* that mediate the interactions between them. The creation and monitoring of these contracts is the responsibility of *organiser*-roles. An organizer role is also responsible for controlling the bindings between the functional roles and the players, within the composite under its control.

2.2 Players

Roles are played by *players*. Players contain the logic of how to perform certain works. Role represents the interface of these works to other roles in the system. The system behaviour is defined by the interaction of roles. The work required by a specific role is performed by its player. However, a player does not have any knowledge about the other associated roles. A role can have different player at different times.

2.3 Contracts

A contract represents an association between two roles. It expresses the obligations of the contracted parties to each other. These obligations are, for example, the required minimum performance standard, the price, the quality of service, etc. A contract also monitors to ensure that the obligations are met by both parties during execution time. If there is any breach of a contract, the organiser role will be informed and it will mitigate the problems based on its strategies.

A contract is the binding between two roles. In addition, a contract can express the control relationship between two roles. For example, the contract between Supervisor and Subordinate is very different from the contract between Peer and Peer.

2.4 Control and Communication Acts (CCAs)

CCAs are abstract messages that express the control relationship between the two parties bound in a contract. For example, in a contract between Supervisor and Subordinate, the Supervisor can tell the Subordinate to do work, but the Subordinate cannot tell the Supervisor to do work. Such method call from Subordinate to Supervisor will be treated as an invalid CCA and will be prevented by the ROAD framework.

Method calls between two uncontracted roles are also considered as invalid CCA and will be prevented.

A transaction instance performed under a term of the contract can be viewed as a sequence of CCAs that are abstractions of the actual underlying messages or method invocations. For example, roles A and B are bound to a contract. A valid transaction between A and B would start by A calling B's **do_work()** method, then B responses by calling A's **inform_workdone()** method. Any other sequence of method calling between A and B will be considered as invalid CCA sequence and will be prevented. However, the current ROAD implementation does not support this feature, it will be addressed at the later stage.

2.5 Composites

A set of functional roles that perform a definable domain functions is called a "self-managed composite". Within each composite, there is an organiser role that is responsible for creating, monitoring and controlling the contracts between functional roles.

A self-managed composite is often part of higher-level composites. A role-based organisation is built from a recursive structure of self-managed composites, in order words it is a hierarchical composition.

Within a composite, there are many roles that interact to one another in order to perform the composite's functionalities. A message allocation mechanism is needed to pass on the incoming requests to their corresponding roles.

2.6 Adaptive Behaviour

As the environment changes or new requirements are introduced, the adaptive system must have the ability to change its internal composition in order to respond to those changes. The ROAD design provides this ability as described below.

- Roles and players are independent entities. Roles can exist independently of players but roles rely on players to execute their functions. A role is not executable if there are no players defined for the roles. A role and player are dynamically bound to each other by the organiser creating mutual references between them. Roles can be filled at different times by different players. Players can be changed depending on the performance requirements of the role and the capability/availability of the players. At runtime, new players can be introduced to the system. In the case of a player is not performing well, the organiser will replace it with a more capable player if such player exists.

- To meet the new requirements, the organiser role can create new roles, create new contracts between roles or revoke the existing contracts, thus changing the internal structure of a composite. The level of intelligence and capability of the organiser in controlling and managing a composite depends on the player playing this organiser role. This player can be changed dynamically. Examples of organiser players are humans using a user interface, artificial intelligence systems, etc. The organiser player represents the mitigation strategy that the current organiser utilises. Strategy can be changed at run time by changing the corresponding organiser player. If the reconfiguration is beyond the current capability of the organiser, the problem will be escalated to the composite at the next higher level.

3 Scope

3.1 Initial Scope

The initial scope of the Research Project includes the following:

1. Complete the implementation of the ROAD framework as described in [1,4] to include the ability to monitor the performance and to mitigate the under-performance by dynamically creating new roles, creating contracts between roles, detaching a player from a role and attaching another player into that role.
2. A complete test program that demonstrates the adaptive capability of ROAD framework.
3. A visual program that allows a user defines roles, associates roles with pre-defined contracts, sets performance requirements to the simulation, and runs the simulation. During run time, the visual program will highlight which contracts are in breach and show the new reconfiguration to demonstrate the adaptive capability of ROAD framework.

3.2 Revised Scope

Similar to many other software engineering projects, implementation of a conceptual design can reveal some problems that were not thought of during the design time. This implementation necessitated specifying the dynamics of the interactions between composites, roles and players within the ROAD framework. In particular, we needed to detail how interactions could be controlled and measured for different types of synchronisation within a ROAD structure. These problems are discussed in section 4 [Issues and Solutions]. A significant amount of time has been spent to resolve these problems.

As better understanding of ROAD framework has been gained, more specific requirements for the test program was plotted out.

The visual program was suggested to be implemented by using Eclipse RCP (Rich Client Platform). However, the Eclipse RCP is quite complex and it is not straight forward to be applied in such a short time frame.

Due to the above reasons and with time constraints, the scope was revised as follow:

1. Develop a prototype implementation the ROAD framework to cope with various styles of transactions (synchronous, asynchronous, etc.) and style of control (push and pull). Document the assumptions and limitations of the prototype.
2. The test program demonstrating the adaptive capability has to show the ability to detach and reattach players, create new roles and binding between roles at runtime to maximise the performance of a composite.
3. It is decided to leave the visual program to a later stage. Recently, we have made some progress in developing the Visual Programming Environment with the collaboration of John Grundy, University of Auckland. However, the visual program is not included this report.

4 Issues and Solutions

In this section we discuss the issues that arose in implementing transactions in ROAD, and give a brief rationale of the design solutions that were selected. These issues include message routing, work allocation (i.e. request message routing) to roles of the same type, queuing of messages, defining performance measurement points for transactions for various synchronisation types, and the implementation of push and pull models of work allocation.

4.1 Message Routing

The focus of this report is the problems encountered in implementing different types of transaction in the ROAD framework. From this perspective, the ROAD framework can be viewed as a *message routing system*, where messages are passed between players via their roles. Players in a ROAD system are “structure shy” [7]. They have no understanding of the organisational structure in which they participate, but rather only talk to their roles. It is the roles that are aware of the contracts that bind them to other roles, and thus it is the roles responsibility to pass outgoing messages (received from the player) to the appropriate party. In Figure 1 below, Role A receives outgoing messages from its player (Player 2), and must know to which role (Role B or Role D) this message must be forwarded.

On the other hand, roles forward *all* incoming messages to their players. If that player is a composite (as is the case with Player 2 in the figure below) the composite needs to *delegate* the incoming message to the appropriate role.

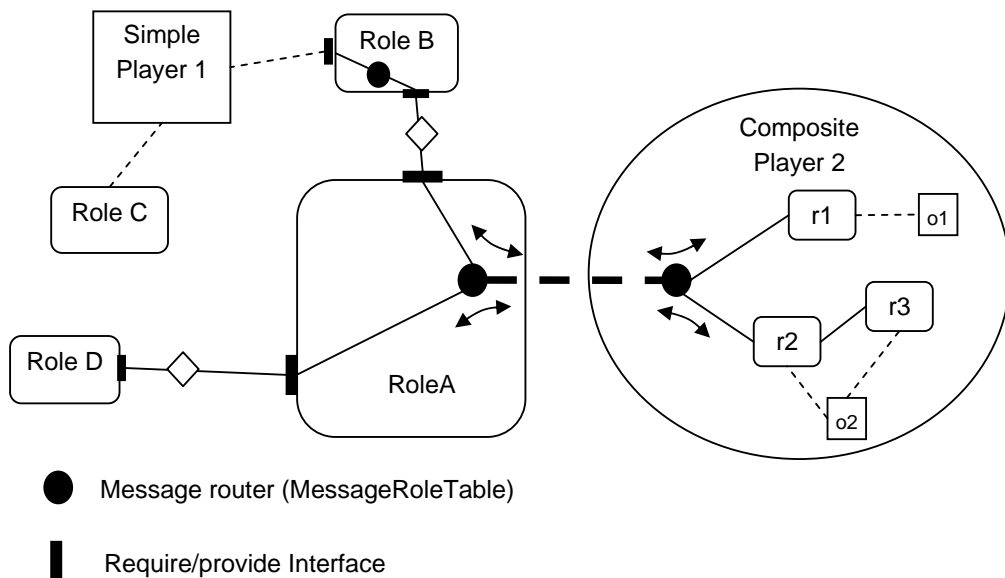


Figure 1 Message routing in a ROAD structure

There are therefore, two places where message routing needs to be performed:

- In Composite: the composite is a collection of roles that performs some given domain functions. The message allocation routes in-coming messages to the appropriate roles inside the composite.
- In Role: a role can be contracted with multiple roles that are capable of performing a given request, the message allocation routes out-going messages to the most suitable role which will receive the request.

Routing in both roles and composites has been implemented using a MessageRoleTable. A MessageRoleTable is an object that has a table of records. MessageRoleTables have a getAllocatedRole() method that returns a reference to the appropriate role given a message signature. Each record in a MessageRoleTable has the message's signature as well as role(s) that can handle the message. Role and Composite will use this table to route the messages to the appropriate role. The entries in the MessageRoleTable are maintained by the composite's organiser.

4.2 Work Allocation

Work allocation is the routing of a request (a DO CCA) to an obligated party. A special case of message routing within a composite is where there are multiple roles of the same type, and the composite must select a role to 'push' the message to, depending on the role-player's capacity, availability etc. In Figure 1 above, inside the composite Player 2, role instances R1 and R2 might be of the same type, and therefore capable of handling the same types of messages.

One approach is to have the composite itself perform work allocation, the message role table can be extended so that there is a one-to-many relationship between messages and roles, i.e. one message type can be allocated to a number of role instances. At runtime the actual allocation of the message is determined by an *allocation scheme* (e.g. Round-robin), which the router in the MessageRoleTable uses. The name of this scheme is record as an extra field in the MessageRoleTable.

An alternative approach is for a special role (e.g. a Foreman role) within the composite that performs the work allocation. This role initial receives all messages of a given type, then uses an allocation scheme to allocate those message to the various roles to the 'worker' roles that are contract to it. Again, this allocator role uses MessageRoleTable to keep track of the other roles to which it is contracted. However, this approach would restrict the reuse of Foreman in other contexts.

4.3 Message Queues in Roles

A role is a "position" to be filled by a player. Roles may be temporarily unassigned to players. If this is the case, there must be some forms of queue to store the incoming requests. Once a player is assigned to a role, this player will then execute the requests that previously piled up.

Queue can be implemented in a contract or in a role. A role can be involved in different contracts. If queue is implemented in a contract, role's player has to access to different queues in different contracts. This will have some problems of which queue it will access first, or it can

alternatively access queues in a round-robin fashion. Even with the alternative round-robin method, it is not intuitive and logical to do so. Further more, there are two way messaging between two contracted roles, from A to B and from B to A. So there must be two queues in one contract. In contrast, if queue is implemented in a role, role's player can retrieve messages in FIFO (First In First Out) fashion, or based on a priority associated with each message in the queue. There will be only one queue in each role, thus resulting in a simpler and more logical solution.

4.4 Synchronisation Types and Performance Measurement

In order to be a complete and robust framework, ROAD will need to address different aspects of non-functional requirements of the system, namely security, reliability, performance, etc. Most of the times, these requirements are conflicting and components can only partially conform to them. For example, a component can have strong security but slow performance. A “smart” utility function which can consolidate those conflicting requirements is desirable.

As a starting point, ROAD framework only supports the performance measurement. “Performance” can mean different things and can be measured by different methods, here we only restrict “performance” in terms of time taken (in milliseconds) to complete a request.

The challenge is to correctly calculate the performance of a transaction in the presence of the queuing system. The queuing system needs to address the different synchronisation type at role level, namely one-way, synchronous, asynchronous and deferred synchronous.

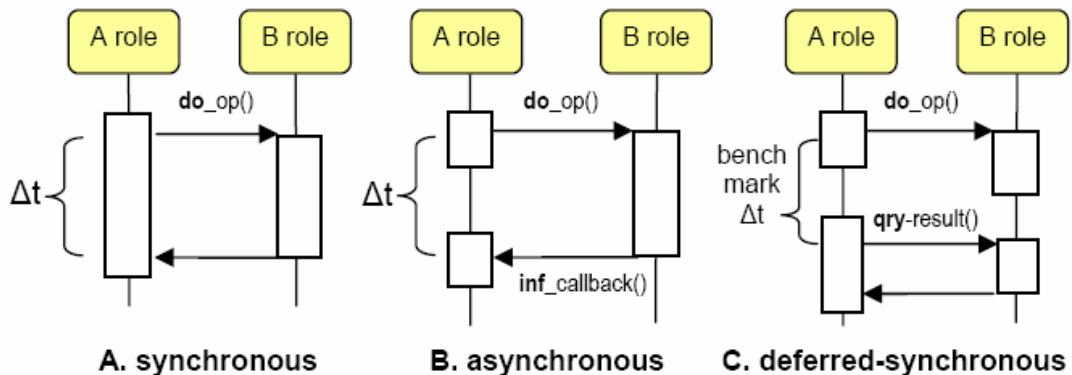


Figure 2: Different Synchronisation Types (from [1])

- With one-way messaging, the result of the transaction is of no interest to the caller. Hence the performance is irrelevant in this case.
- With synchronous messaging, the caller is blocked during the transaction. If the player is not present, the caller(s) can wait until the player is assigned to the role and perform the action. If the caller(s) cannot afford to wait for a long time, a time-out can be implemented. Another possibility is to throw an exception. In any of these cases, a message queue is not required. To measure the performance, we can capture and measure the necessary information by intercepting messages between roles and monitoring the CCAs of those messages. This is implemented by using AspectJ and Association Aspect's point-cuts. An example is to measure the start time of the call at the *before point-cut*, and measure the

finish time of the call at the *after point-cut*. Hence the execution time of the call is the subtraction of finish time from start time. This method makes use of point-cuts to dynamically intercept messages, therefore the code for management function and domain function of a role do not cluster together. The management function code is only coded once, thus facilitate reuse of code.

- With deferred synchronous messaging, the caller will query the result from its obligated party (the role that provides the service to caller). However, if the result is not available yet, the caller has to perform another query some times in the future (i.e. polling). The performance of this transaction can be relatively defined from the time when the request is sent to the time when a successful query is carried out. The term “relatively” is used since the actual execution time is always less than the time calculated by this method because of the pending time until a query is performed. Similar to the “synchronous messaging case”, the implementation is done by using Association Aspect’s point-cuts. It records the start time at the *before point-cut* of the request, and the finish time at the *after point-cut* of the query. Based on the return values, it will ignore the query when the result is not available yet. This approach suffers the same limitation as the “synchronous messaging” case (i.e. reliance on method calls), and as stated previously, it can only “relatively” determine the performance of the transaction. The ROAD framework implementation at this stage does not support deferred synchronous messaging yet, but the extension can be done easily.
- With asynchronous messaging, the obligated party will perform a call back to the caller once the execution is completed. The implementation can be done by recording the start time at the *before point-cut* of the request, and the finish time at the *before point-cut* of the call back. Suppose role A places a number of asynchronous requests on role B, it may be very important to match the correct call back to its corresponding request. The ROAD framework implementation can handle this case correctly based on the `AsynchronousTransaction` which stores the signature of the request and its corresponding call back. In the *before point-cut* of the call back, we can match to the correct request and finalise this transaction by calculating the performance. For requests coming from the same role and of the same request type (i.e. method signature), it is assumed that these messages are processed in FIFO (First In First Out) manner. It is not likely that the requests coming from the same role and of the same request type can have different priority. However, to make the framework more robust, the transaction ID can be used to correctly match a call back to its corresponding request among the pending transactions. Transaction ID is not supported by current ROAD implementation.

4.5 Pull vs. Push Models

The above solution is based on a Push model. In Push model, the message queue resides in the obligated party (the role that provides the service to client role). The client makes a request by calling the obligated party’s method, in other words, it pushes the messages to the obligated party (hence the name Push model). However, there is one problem with this model. To mitigate the underperformance, the organiser is capable of creating roles to utilise the existing players. Suppose role A places a large amount of requests to role B. As role B is degrading and does not meet the requirement, a new role (e.g. role C) is created by the organiser. Some of the messages previously allocated to role B may need to be transferred to role C. However, this

transfer cannot be done easily as taking messages out from B requires some cleaning up actions for the transaction to be maintained correctly.

One solution to the above problem is the Pull model. In Pull model, the message queue resides in the client role (i.e. the caller). In this model, interaction is demand-driven. If the client wants to make a request to the obligated party, it simply places the request message into the queue. It is in fact the out-going queue of the client role. The registered obligated role will pop the messages out and process them. Once it finishes processing a message, it will come back to the message queue to get the next message. If there are any newly created roles, these new roles will participate in the same process. Since the message queue resides in the client role, the transfer of messages from role B to the new role C is irrelevant.

However, there are some problems encountered while attempting to implement this Pull model:

- If there are multiple roles retrieving messages from the queue, performing the work then calling back to the client once they finish, it is not possible to correctly match the call back to the original request. In other words, the identity of the message is lost. One way to keep track of this identity is by associated a transaction ID to each message.
- Messages with parameters are hard to handle correctly. For example, role A has a message of an order of 1000 items and another message with an order of 10 items. A role B which is bound to a slow player might get a 1000 item order while role C which is bound to a very fast player might get a 10 item order. In this case, the capability of the system is not utilised. A possible solution is to break any given message into atomic messages (i.e. message of an order of 1 item). However, not all kind of messages can be broken down to atomic messages.
- The performance is calculated based on the method signature point-cut (i.e. role A calls a method in role B). Now if role A just puts the messages into its out-going queue, there will be no method calling to role B. Therefore, the performance calculation cannot be carried out based on the message intercepting method.
- There must be different out-going queues for different role types. For example, role A is contracted to role B to make item b, it is also contracted to role C to make item c. Clearly, there should be two out-going queues, one for the messages of item b orders, and another one for the messages of item c orders. Another solution is to have only one out-going queue in A, when there is messages in the queue, it will notify the registered roles (i.e. role B and C), B and C will in turn lock the queue, examine and take only messages assigned to them. This solution will result overhead in notification and locking.
- If there is an option between Push or Pull model, message distribution can be a problem. For example, role A has requests that will be performed by role B, C, and D. Role B and C are using Pull model, whereas role D is using Push model. How can we distribute the messages to them?

With the complexity involved in the implementation of the Pull model, it is decided to leave the Pull model to a later stage due to time constraints.

5 Results

5.1 State of ROAD Framework

The ROAD Framework implementation has achieved most functionality discussed in [1]. In particular, the following is implemented:

- **Role:** different players can be assigned to a specific role dynamically. Role can interact with each other. Push model of messaging is implemented for asynchronous synchronisation type messages. Each role has its own thread that will process the asynchronous messages and perform the call backs to the client. Message allocation is implemented to route the incoming and out-going messages. Different allocation schemes (e.g. round robin, etc) can be used and changed at run time.
- **Player:** player can be assigned to role(s). Player contains the logic of how to perform a particular domain function. Player has a claimed performance which can be used by organiser to determine which player is the best to fill in a particular role.
- **Contract:** contract is the binding between two roles. It defines the requirements and obligations between two roles in a number of terms. The contract then monitors the performance of the transactions between two roles. It notifies the organiser when underperformance or in breach of contract is detected.
- **Organiser:** organiser can create new roles, new contracts between roles. Organiser can mitigate the underperformance or in breach of contract. The level of intelligence of the organiser in mitigation can be varied easily by changing the organiser player that is bound to this organiser role. In order to have the ability to create new roles, the organiser has a reference to the IRoleFactory interface.
- **Composite:** composite contains a set of roles that together performs some given domain functions. Message allocation routes the incoming messages to their corresponding roles.

The ROAD Framework simplified UML diagram:

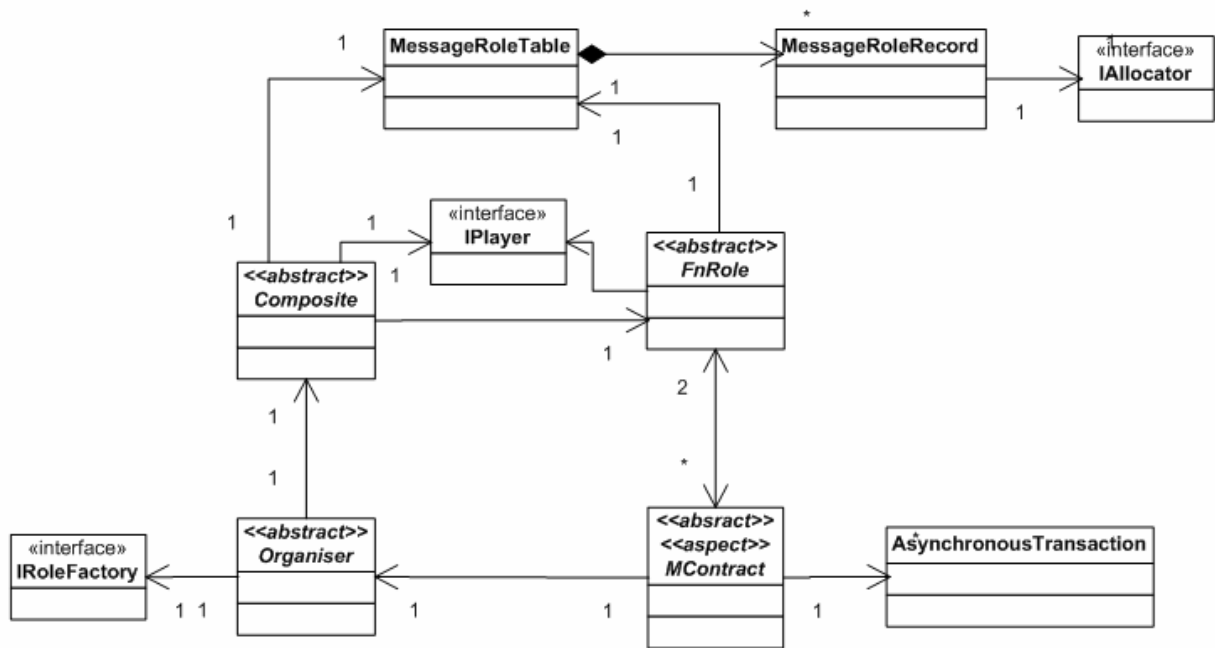


Figure 3: ROAD Framework Simplified UML Diagram

5.2 ROAD Domain Specific Application

The ROAD framework was used to implement a domain specific organisation. The domain used in testing was an imaginary widget making department. Players of varying capability were simulated in order to demonstrate ROAD's adaptive behaviour.

The Widget Department is responsible for making widget from the order of the Widget Maker. Inside the Widget Department, there are roles that are responsible to make thingies then assemble those thingies to produce a widget. There is certain level of performance that the Widget Department must meet according to the terms of the contract C1. During execution, degradation is detected in thingy production, thus the Widget Department has to reconfigure its internal structure by changing thingy maker players, or changing role structure in order to utilise available players. This adaptive behaviour recovers the Widget Department performance; hence increase its viability in its environment.

5.2.1 Widget Making Department

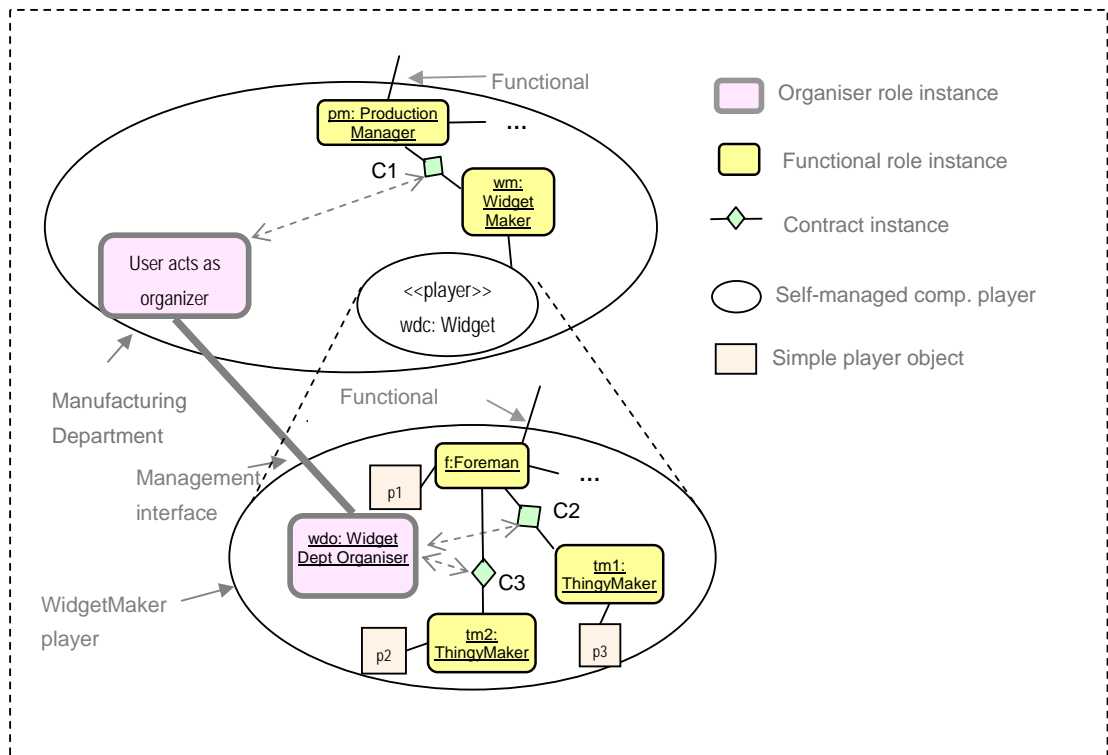


Figure 4: Widget Making Department (from [1])

In this domain, there are two roles at the top level composite: ProductionManager and WidgetMaker. A lower level composite, WidgetDeptComposite, acts as a player of WidgetMaker role. Inside this composite, there are contracts between Foreman and ThingyMaker roles. The WidgetDeptOrganiser set some pre-defined requirements in contracts between Foreman and ThingyMaker roles.

To make widgets, the request message is sent to ProductionManager. Its player contains specific logic of how to obtain the number of widgets requested. In our application, the number of widgets requested is obtained by asking the user. This method can be changed easily to other methods (e.g. by asking the higher level composite) by changing different players.

By using the out-going message allocation, the ProductionManager then requests the WidgetMaker to make a given number of widgets. This request is passed from WidgetMaker role to its player which is a composite WidgetDeptComposite.

In WidgetDeptComposite, the in-coming request is routed to Foreman. Widget in this simple application is made up of only a thingy. So Foreman will route its out-going messages to the ThingyMaker.

5.2.2 UML Class Diagram of Widget Making Department

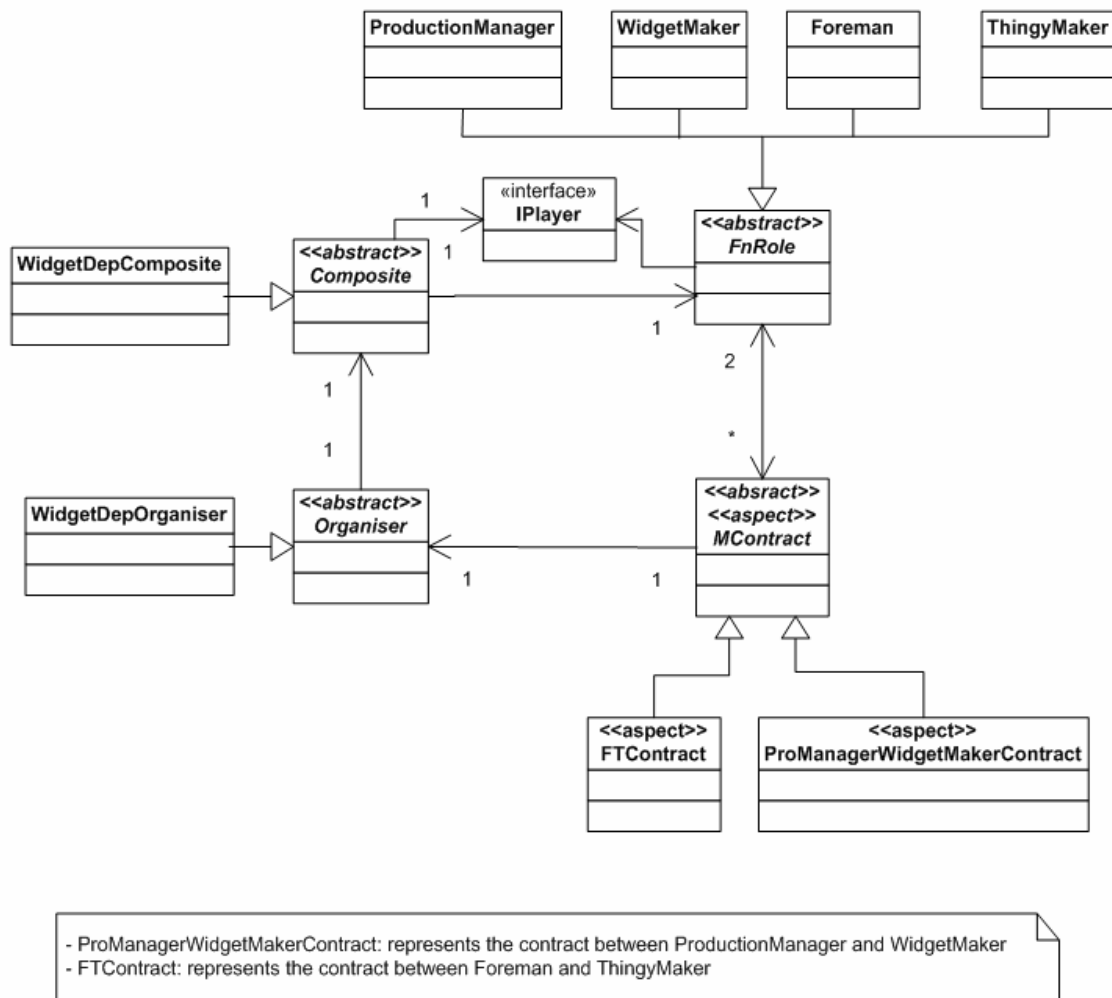


Figure 5: Widget Making Department UML Simplified Class Diagram

5.2.3 Test Harness

Please refer to section Appendix A [Test Harness Code] for the complete code of the test harness.

In the test harness, from line 21 to line 47, the necessary roles, composite and organiser are created. SkillfulEmployee objects are players that can be assigned to both Foreman role and ThingyMaker role. LazyEmployee objects are players that can only be assigned to ThingyMaker role. From line 54 to line 80, the program sets up initial players and adds the roles to the composite. The initial players of the roles can be changed later during execution time if the players do not meet the performance requirements. A SkillfulEmployee object produces a thingy in 20 ms. Whereas a LazyEmployee object produces a thingy in 10 ms, however after each thingy is made, its performance increases by 20 ms (i.e. 30 ms then 50 ms, and so on). The cap performance is 100 ms,

after which it remains unchanged. The time taken to make a thingy is arbitrarily chosen and it is implemented by Thread.sleep() method.

From line 82 to line 94, the order of widget is placed by calling the method do_placeOrderWidgets() of ProductionManager. After taking the input of the number of widgets required from the user, the ProductionManager sends request to WidgetMaker. However, as the contract between ProductionManager and WidgetMaker has not been created yet, the framework will prevent the invalid CCA between two uncontracted roles.

From line 96 to 113, the contract ProManagerWidgetMakerContract is created between ProductionManager and WidgetMaker. Again, we place an order of widget by calling do_placeOrderWidgets() of ProductionManager. This time, the message from ProductionManager to WidgetMaker is not blocked as it is a valid CCA. However, as the message flows through the organisation, Foreman sends messages to ThingyMaker for thingies to be made. These messages got blocked since Foreman and ThingyMaker are uncontracted roles, and the contract between them has not been created yet.

From line 114 to 132, the contract between Foreman and ThingyMaker is created. Order of widgets can be fulfilled now.

5.2.4 Test Harness Output

For the output of the test harness, please refer to section Appendix B [Test Harness Output].

From line 1 to 7, it informs that inside the WidgetDepComposite object, there are three ThingyMaker players: foremanPlayer, badThingyMakerPlayer, and goodThingyMakerPlayer. Since SkillfulEmployee objects can be assigned as Foreman player or ThingyMaker player, and a player can play multiple roles at the same time, therefore the “foremanPlayer” can be considered as a potential ThingyMaker player. Initially, there are one Foreman object, and one ThingyMaker object. foremanPlayer is bound to Foreman f, badThingyMakerPlayer is bound to ThingyMaker t. This configuration is depicted by the following figure.

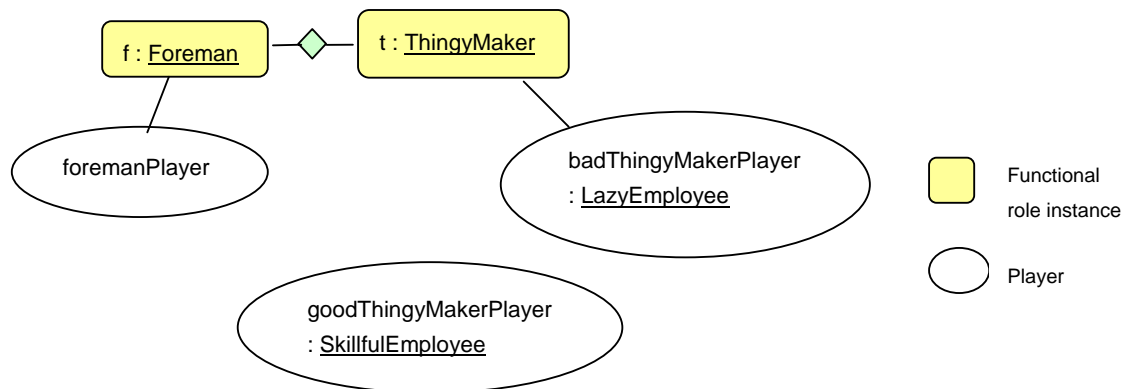


Figure 6: Initial Components Inside WidgetDepComposite Instance

From line 9 to 14, it informs that the invalid CCA between uncontracted ProductionManager and WidgetMaker is prevented. From line 16 to 19, it shows that the contract between ProductionManager and WidgetMaker is created with its corresponding two terms.

From line 22 to 28, it show that invalid CCA between uncontracted Foreman and ThingyMaker is prevented. From line 31 to 35, it shows that the contract between Foreman and ThingyMaker is created with its corresponding two terms.

From line 38 to the end, the order of 15 widgets is processed. As widget is simply made of only one thingy, we only show thingy in the output. The requests from Foreman to ThingyMaker to make thingies are asynchronous requests. For the order of 15 widgets, Foreman will place 15 requests to the ThingyMaker, each request is for 1 thingy.

There are two threads running in the background, one is the main thread, the other thread is the thread inside the ThingyMaker role processing the asynchronous messages. Line 47 shows that the main thread pushes message into ThingyMaker role's message queue. The message processing thread then pops the message out and processes it. Line 50 shows that the first thingy is made by badThingyMakerPlayer in 10 ms. The elapsed time on line 53 is slightly more than 10 ms (it shows 15 ms) because of the pending time while the message is inside the queue. Line 56, it informs the user that a thingy is made.

Line 64 shows that the badThingyMakerPlayer is degrading. Now it makes a thingy in 30 ms. Line 81 shows that the time taken for the request to be processed including the time pending while inside the queue is 94 ms and it is in breach of the contract requirements. Line 89 shows that the organiser replaces the badThingyMakerPlayer by a better player. In this case, the better player is either foremanPlayer or goodThingyMakerPlayer as they both have the performance of 20 ms. The choice is arbitrary and foremanPlayer is chosen.

The configuration of the WidgetDepComposite instance is changed dynamically. The foremanPlayer is now assigned to both Foreman and ThingyMaker roles.

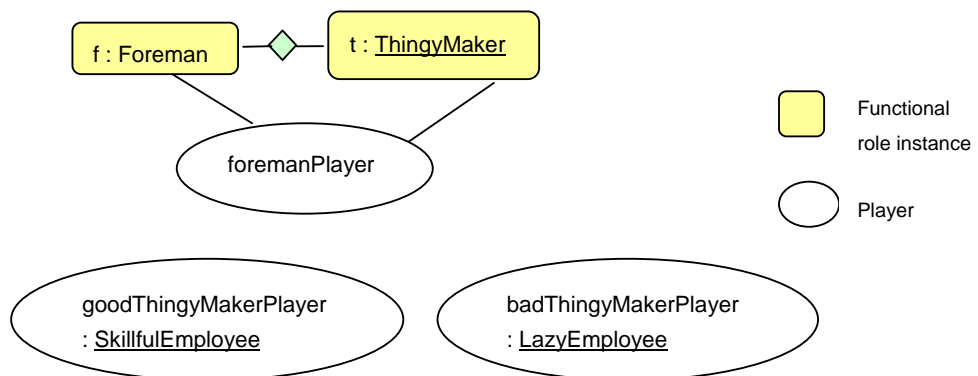


Figure 7: The Configuration of WidgetDepComposite Is Changed Dynamically

Line 98 shows that the next thingy is made of this newly assigned ThingyMaker player. It made a thingy in 20 ms. However, including the time in the pending queue, the elapsed time for this particular request is 93 ms shown on line 101. Since we just changed the player, the average performance of this new player is calculated based only on the new player's performance. Line 102 shows that the average is correctly calculated to 93.0 ms. This performance once again is in breach of the contract requirements. However, there is no better player. The organiser detects that there is unused resources in the composite (in this case, goodThingyMakerPlayer and badThingyMakerPlayer are unused). To minimise the change in the configuration, it tries to utilise one player at the time. To utilise a ThingyMaker player, it has to create a new ThingyMaker role, and create a contract between this new ThingyMaker role and the existing Foreman. It then binds the best available player (in this case, that is goodThingyMakerPlayer with performance of 20 ms) to the newly created ThingyMaker role.

The new configuration is as follow.

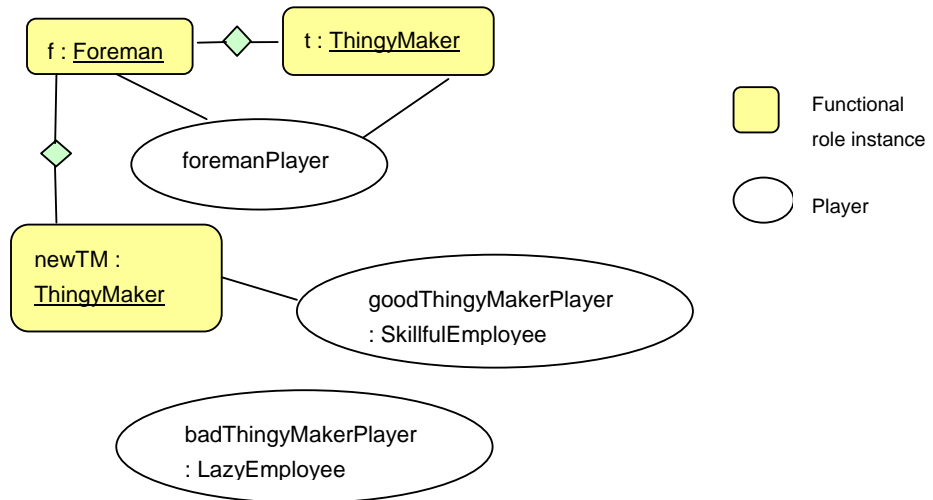


Figure 8: Another Configuration of WidgetDepComposite Instance At Run Time

Since there is a message processing thread inside each ThingyMaker role instance, there are three threads running at this moment. One is the main thread, and the other two are threads inside two ThingyMaker role instances. The main thread is putting messages from Foreman to two ThingyMaker instances, and the two threads are processing messages. These threads output to the screen in no particular order. This is the reason why the output is quite hard to follow.

Line 153 shows that the `goodThingyMakerPlayer` is actually participating in creating thingies. The elapsed time and average are calculated separately for each of the players.

Line 279 shows that the last thingy is made. The program does not terminate, since the main thread finishes but the two threads in ThingyMaker instances are still running and waiting for new messages in their queues.

5.2.5 Conclusion

The test program has shown that the ROAD framework is adaptive by changing its internal structure to utilise its resources to meet the requirements.

It demonstrated the ability of calculating and monitoring the performance of players. It shows the ability of organiser in swapping players when the contract requirements is in breach, and creating new role instances to utilise the available resources inside the composite.

6 Future Work

The future work to extend and make the framework more robust is to implement the transaction ID, the Pull model, and define a more concrete Signature as discussed in section 4 [Issues and Solutions].

The specific configuration of roles, players, composites, organisers to match a given domain could be done via loading from configuration files in XML format. Defining this format and the loading functionality are desirable. Then the visual tool can be used to define the contract between roles, binding between roles and players.

The ROAD framework can also be extended to be used in Web Services area.

References

- [1] A. Colman and J. Han, Coordination Systems in Role-based Adaptive Software, in: Proceedings of the 7th International Conference on Coordination Models and Languages (COORD 2005), LNCS 3454, Namur, Belgium, 2005, 63-78.
- [2] Eclipse Foundation, *AspectJ* <http://eclipse.org/aspectj/>, 2004, *last accessed 7 Oct 2004*
- [3] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya, Association Aspects, in: Proceedings of the Aspect-Oriented Software Development '04, Lancaster U.K., 2004, 16-25.
- [4] A. Colman and J. Han, Using Associations Aspects to Implement Organisational Contracts, in: Proceedings of the 1st International Workshop on Coordination and Organisation (CoOrg 2005), Namur, Belgium, 2005
- [5] A. Colman and J. Han, An organisational approach to building adaptive service-oriented systems, in: Proceeding of First International Workshop on Engineering Service Compositions, WESC'05, in IBM Research Report RC23821, Amsterdam, The Netherlands, 2005
- [6] A. Colman and J. Han, Operational management contracts for adaptive software organisation, in: Proceedings of the Australian Software Engineering Conference (ASWEC 2005), Brisbane, Australia, 2005, 170-179.
- [7] Lieberherr, K. J. *Adaptive object-oriented software the Demeter Method with propagation patterns*, Boston: PWS Pub. Co, 1996.

Appendices

Appendix A – Test Harness Code

```
1 package widgetOrg;
2
3 import mContract.Composite;
4 import mContract.Organiser;
5 import widgets.Foreman;
6 import widgets.LazyEmployee;
7 import widgets.Manager;
8 import widgets.NonFnRole;
9 import widgets.ProductionManager;
10 import widgets.SkillfulEmployee;
11 import widgets.ThingyMaker;
12 import widgets.WidgetMaker;
13
14 /**
15  * A test program to test the functionality of a composite, and the abilities
16  * to swap players and to create new role at run time of organiser.
17  *
18  * @author Alan Colman
19  * @author Linh Duy Pham
20  */
21 public class TestComposite
22 {
23     public static void main(String[] args)
24     {
25         ProductionManager pm = new ProductionManager("Production Manager");
26         WidgetMaker wm = new WidgetMaker("Widget Maker");
27
28         //player for ProductionManager
29         Manager manager = new Manager("Manager");
30
31         // Organiser setup
32         WidgetDepOrganiserPlayer orgPlayer = new WidgetDepOrganiserPlayer();
33         Organiser org = new WidgetDepOrganiser(new WidgetDepRoleFactory());
34         org.setPlayer(orgPlayer);
35
36         // Player for WidgetMaker --> create the WidgetDepComposite
37         Composite widgetDepComposite = new WidgetDepComposite();
38         org.setComposite(widgetDepComposite);
39
40         //create ThingyMaker and Foreman
41         ThingyMaker t = new ThingyMaker("Thingy Maker");
42         Foreman f = new Foreman("Foreman");
43
```

```

44 //create other players
45 SkillfulEmployee foremanPlayer = new SkillfulEmployee("Foreman/Thingy Player");
46 SkillfulEmployee goodThingyMakerPlayer = new SkillfulEmployee("Skillful Thingy
Maker");
47 LazyEmployee badThingyMakerPlayer = new LazyEmployee("Lazy Thingy Maker");
48
49 System.out.println("\nThere are three ThingyMaker players created.");
50 System.out.println("Foreman/Thingy Player: performance 20 ms");
51 System.out.println("Skillful Thingy Maker: performance 20 ms");
52 System.out.println("Lazy Thingy Maker: performance at start is 10 ms, increase
by 20 ms every time a thingy is made, with cap of 100 ms");
53
54 // Setup Initial Players
55 try
56 {
57 //ProductionManager
58 pm.setPlayer(manager);
59
60 //WidgetMaker
61 wm.setPlayer(widgetDepComposite);
62
63 //Foreman and ThingyMaker
64 f.setPlayer(foremanPlayer);
65
66 System.out.println("\nStart up program with 1 ThingyMaker role, Lazy Thingy
Maker is the initial player.");
67 t.setPlayer(badThingyMakerPlayer);
68 }
69 catch (Exception e)
70 {
71 System.out.println(e.getMessage());
72 }
73
74 // add Roles and Players to composite
75 widgetDepComposite.addRole(f);
76 widgetDepComposite.addRole(t);
77
78 widgetDepComposite.addPlayer(foremanPlayer);
79 widgetDepComposite.addPlayer(goodThingyMakerPlayer);
80 widgetDepComposite.addPlayer(badThingyMakerPlayer);
81
82 System.out.println("\n\n-----");
83 System.out.println("---- Before contract between Production Manager and
WidgetMaker is created ----");
84 System.out.println("TEST: Should have error non contracted between
ProductionManager and WidgetMaker");
85

```

```

86     try
87     {
88         pm.do_placeOrderWidgets();
89     }
90     catch (Exception e)
91     {
92         System.out.println(e.getMessage());
93         e.printStackTrace();
94     }
95
96     System.out.println("\n\n-----");
97     System.out.println("---- Creating contract between ProductionManager and
WidgetMaker ----");
98     // create contract
99     ProManagerWidgetMakerContract      contract      =      new
ProManagerWidgetMakerContract(pm, wm);
100
101     System.out.println("\n\n-----");
102     System.out.println("---- Before contract between Foreman and ThingyMaker is
created ----");
103     System.out.println("TEST: Should have error non contracted between Foreman
and
ThingyMaker");
104
105     try
106     {
107         pm.do_placeOrderWidgets();
108     }
109     catch (Exception e)
110     {
111         System.out.println(e.getMessage());
112         e.printStackTrace();
113     }
114
115     System.out.println("\n\n-----");
116     System.out.println("---- Create contract between Foreman and ThingyMaker ----");
117     org.createContract(f, t);
118
119     System.out.println("\n\n-----");
120     System.out.println("---- Now place an order of widget ----");
121     System.out.println("TEST: Should produce widgets");
122
123     try
124     {
125         pm.do_placeOrderWidgets();
126     }
127     catch (Exception e)

```

```

128     {
129         System.out.println(e.getMessage());
130         e.printStackTrace();
131     }
132 }
133
134 }

```

Appendix B – Test Harness Output

```

1 There are three ThingyMaker players created.
2 Foreman/Thingy Player: performance 20 ms
3 Skillful Thingy Maker: performance 20 ms
4 Lazy Thingy Maker: performance at start is 10 ms, increase by 20 ms every time a thingy
  is made, with cap of 100 ms
5
6 Start up program with 1 ThingyMaker role, Lazy Thingy Maker is the initial player.
7
8
9 -----
10 ---- Before contract between Production Manager and WidgetMaker is created ----
11 TEST: Should have error non contracted between ProductionManager and WidgetMaker
12 To user: Enter number of widgets required: 15
13 X--X CCA call from uncontracted functional role: call(void widgets.WidgetMaker.
  do_makeWidget(int))
14
15
16 -----
17 ---- Creating contract between ProductionManager and WidgetMaker ----
18 do_makeWidget term added to contract
19 qry_widgetOrder term added to contract
20
21
22 -----
23 ---- Before contract between Foreman and ThingyMaker is created ----
24 TEST: Should have error non contracted between Foreman and ThingyMaker
25 To user: Enter number of widgets required: 15
26 ---> before a1 do AtoB : call(void widgets.WidgetMaker.do_makeWidget(int)) - Calculate
  Start time.
27 ---> after a0 error: call(void widgets.WidgetMaker.do_makeWidget(int))
28 X--X CCA call from uncontracted functional role: call(void widgets.ThingyMaker.
  do_makeThingy())
29
30
31 -----
32 ---- Create contract between Foreman and ThingyMaker ----

```

```

33 Utility: FTUtility: thingiesPerSec0
34 do_makeThingy term added to contract
35 inf_thingyMade term added to contract
36
37
38 -----
39 ---- Now place an order of widget ----
40 TEST: Should produce widgets
41 To user: Enter number of widgets required: 15
42 ---> before a1 do AtoB : call(void widgets.WidgetMaker.do_makeWidget(int)) - Calculate
Start time.
43
44 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
Start time.
45 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
46 signature: do_makeThingy
47 Message added.
48
49 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
Start time.
50 Thingy made by Lazy Employee named Lazy Thingy Maker, in 10 ms
51
52 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
53 afterUpdate Last elapsedtime 15 msec
54 Moving average is 15.0 msec
55 calculateUtility
56 To user: Thingies are made. Quantity = 1
57 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
58 signature: inf_thingyMade
59 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
60 signature: do_makeThingy
61 Message added.
62
63 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
Start time.
64 Thingy made by Lazy Employee named Lazy Thingy Maker, in 30 ms
65
66 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
67 afterUpdate Last elapsedtime 32 msec
68 Moving average is 23.5 msec
69 calculateUtility
70 To user: Thingies are made. Quantity = 1
71 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
72 signature: inf_thingyMade
73 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
74 signature: do_makeThingy
75 Message added.

```

76
77 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
78 Thingy made by Lazy Employee named Lazy Thingy Maker, in 50 ms
79
80 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
81 afterUpdate Last elapsedtime 94 msec
82 Moving average is 47.0 msec
83 calculateUtility
84 widgets.ThingyMaker:Thingy Maker: do_makeThingy is in breach
85 IN BREACH
86
87 *** In Breach Action
88
89 *** Replaced by a better player
90 To user: Thingies are made. Quantity = 1
91 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
92 signature: inf_thingyMade
93 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
94 signature: do_makeThingy
95 Message added.
96
97 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
98 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
99
100 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
101 afterUpdate Last elapsedtime 93 msec
102 Moving average is 93.0 msec
103 calculateUtility
104 widgets.ThingyMaker:Thingy Maker: do_makeThingy is in breach
105 IN BREACH
106
107 *** In Breach Action
108 Utility: FTUtility: thingiesPerSec0
109 do_makeThingy term added to contract
110 inf_thingyMade term added to contract
111
112 NOTE: New contract is created between object of class widgets.Foreman and object of class widgets.ThingyMaker
113 To user: Thingies are made. Quantity = 1
114 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
115 signature: inf_thingyMade
116 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
117 signature: do_makeThingy
118 Message added.
119

120 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.

121 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

122

123 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())

124 afterUpdate Last elapsedtime 63 msec

125 Moving average is 78.0 msec

126 calculateUtility

127 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming

128 UNDER-PERFORMANCE

129 To user: Thingies are made. Quantity = 1

130 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())

131 signature: inf_thingyMade

132 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())

133 signature: do_makeThingy

134 Message added.

135

136 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.

137 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())

138 signature: do_makeThingy

139 Message added.

140

141 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.

142 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

143

144 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())

145 afterUpdate Last elapsedtime 63 msec

146 Moving average is 73.0 msec

147 calculateUtility

148 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming

149 UNDER-PERFORMANCE

150 To user: Thingies are made. Quantity = 1

151 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())

152 signature: inf_thingyMade

153 Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms

154

155 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())

156 afterUpdate Last elapsedtime 31 msec

157 Moving average is 31.0 msec

158 calculateUtility

159 To user: Thingies are made. Quantity = 1

160 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())

161 signature: inf_thingyMade

162 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())

163 signature: do_makeThingy

164 Message added.
165
166 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
167 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
168 signature: do_makeThingy
169 Message added.
170
171 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
172 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
173
174 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
175 afterUpdate Last elapsedtime 62 msec
176 Moving average is 70.25 msec
177 calculateUtility
178 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
179 UNDER-PERFORMANCE
180 To user: Thingies are made. Quantity = 1
181 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
182 signature: inf_thingyMade
183 Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms
184
185 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
186 afterUpdate Last elapsedtime 31 msec
187 Moving average is 31.0 msec
188 calculateUtility
189 To user: Thingies are made. Quantity = 1
190 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
191 signature: inf_thingyMade
192 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
193 signature: do_makeThingy
194 Message added.
195
196 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
197 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
198 signature: do_makeThingy
199 Message added.
200
201 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate Start time.
202 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
203
204 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
205 afterUpdate Last elapsedtime 62 msec
206 Moving average is 68.6 msec

207 calculateUtility
 208 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
 209 UNDER-PERFORMANCE
 210 To user: Thingies are made. Quantity = 1
 211 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
 212 signature: inf_thingyMade
 213 Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms
 214
 215 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
 216 afterUpdate Last elapsedtime 31 msec
 217 Moving average is 31.0 msec
 218 calculateUtility
 219 To user: Thingies are made. Quantity = 1
 220 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
 221 signature: inf_thingyMade
 222 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
 223 signature: do_makeThingy
 224 Message added.
 225
 226 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
 Start time.
 227 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
 228 signature: do_makeThingy
 229 Message added.
 230
 231 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
 Start time.
 232 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
 233
 234 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
 235 afterUpdate Last elapsedtime 63 msec
 236 Moving average is 67.66666666666667 msec
 237 calculateUtility
 238 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
 239 UNDER-PERFORMANCE
 240 To user: Thingies are made. Quantity = 1
 241 Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms
 242
 243 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
 244 afterUpdate Last elapsedtime 32 msec
 245 Moving average is 31.25 msec
 246 calculateUtility
 247 To user: Thingies are made. Quantity = 1
 248 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
 249 signature: inf_thingyMade
 250 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
 251 signature: inf_thingyMade

252 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
253 signature: do_makeThingy
254 Message added.
255
256 ---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - Calculate
Start time.
257 ---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
258 signature: do_makeThingy
259 Message added.
260 ---> after a0 : call(void widgets.WidgetMaker.do_makeWidget(int))
261 signature: do_makeWidget
262 Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms
263
264 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
265 afterUpdate Last elapsedtime 31 msec
266 Moving average is 31.2 msec
267 calculateUtility
268 To user: Thingies are made. Quantity = 1
269 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
270 signature: inf_thingyMade
271 Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
272
273 <--- before b1 : call(void widgets.Foreman.inf_thingyMade())
274 afterUpdate Last elapsedtime 63 msec
275 Moving average is 67.0 msec
276 calculateUtility
277 widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
278 UNDER-PERFORMANCE
279 To user: Thingies are made. Quantity = 1
280 <--- after b0 : call(void widgets.Foreman.inf_thingyMade())
281 signature: inf_thingyMade
282