Master Thesis
Software Engineering
Thesis no: MSE-2004-31
May 15

Creating interface-controllers using model driven architecture

Carl Björk
Per Salomonsson

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Contact Information:
Authors:
Carl Björk
Email: pt00cbj@student.bth.se

Per Salomonsson
Email: pt00psa@student.bth.se

External Advisor:
Johan Silvander
Ericsson AB
Address: Ölandsg. 1, Box 518, 371 23 Karlskrona

University Advisor:
Daniel Häggander
TEK

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : http://www.bth.se/tek
Phone : +46 457 38 50 00
Fax : +46 457 271 25

ABSTRACT

In this thesis we will examine a telecom industry case, where combining synchronous and asynchronous interfaces causes problems.
A solution to the problem is being presented in form of an interface controller framework that is based on patterns of common functionality of interface controllers.
The solution is implemented using four different implementation methods (Java, Erlang, XDE, Executable UML), and compared in lines of code, performance and throughput.

Keywords: xUML, Model Driven Architecture, Java, Erlang, XDE, Interface Controllers, synchronous, asynchronous

TABLE OF CONTENT

1 INTRODUCTION

This chapter introduces background and problem description along with the work which will be conducted. Also how the report is structured is presented.

1.1 BACKGROUND AND PROBLEM DESCRIPTION

In the beginnings of software development, the developed software systems were often restricted to use the input and output of the local computer where the software was executed. Today when almost everything is connected to computer networks and the Internet, the software systems of today are not limited to the local computer anymore. Instead software often depends on other systems to perform their tasks and must have means to communicate with these systems.

The communications between software systems are realized by implementing an interface to each communicating end in the systems. The interface describes in detail, the form and structure of the data and how to control the communication of the data passed between the systems. This is required so the systems can interpret the data sent from another system. Different kinds of interfaces exist for different kinds of systems. For example a webserver on the internet has an interface defined by the HTTP standard, which also every web browser understands and because of that, it can interpret the data sent from a webserver and present it as a webpage to the user.

As more advanced software systems emerges, these systems are often required to communicate with multiple different systems to achieve their tasks. Since the other systems were developed and deployed before the development of the newer system, they already have their defined interfaces for communication. Since the different systems could have different types of interfaces, this means that the new systems must be aware of all the different interfaces of the systems it must communicate with. The system under development might have to access data through one type of interface from a specific system and then push this data across another type of interface to another type of system. This part of interface translation could easily become very problematic because of the differences in the design of the interfaces.

For example one system could use an asynchronous interface for communication towards one system and then wants to communicate with another system that uses a synchronous interface. Since the first system uses asynchronous communication, it means that it does not stop and wait for the response from the other system or that it means it expects the response to be sent back immediately. The other system, which uses synchronous communication, expects the other end to wait for the response from itself, when it has completed its processing of the data.

1.2 THE PROBLEM IN TELECOM INDUSTRY

The problems described above are a big issue today in the telecom industry. Since telecom operators want to develop new competing services to its customers, it often involves combining IT and telecom systems together. This can be a problematic matter because of the different design of the interfaces to these systems. For example, many telecom systems use synchronous communication, which will be interfaced by an asynchronous IT system. Telecom systems often receive requests to handle large batch jobs, which are preferable handled in a synchronous manner. Another example could be the need of a transaction-based interface between systems.

Another problem of combining telecom and IT-systems is the fact that many telecom system are also designed to achieve high availability, which means they are often using clusters to ensure availability and for load balancing. The ability to perform communication requests in a transaction based manner is also important in IT-systems. Because of this, telecom systems communicating with the IT-system must be aware of this. The telecom system must for example be able to rollback a failed transaction. The problem is that not many telecom systems are transaction aware.

Today those problems are solved by developing an interface controller for a specific telecom interface. This interface controller is placed in between the telecom system and the IT-system. The interface controller will be aware of the different communication models used for the interfaces and translate requests from one to the other. This includes the transformation of asynchronous requests to synchronous requests and other interface differences between the systems.

The problem with these interface controllers are that every time a new service, new IT-system or a new interface will be integrated, a new interface controller also has to be developed specific for the interface combination. Large parts of the interface controllers have to be rewritten even though they share basically the same functionality.

A way to solve the problems of spending much time developing the interface controllers could be to describe the different patterns of functionality and translations being made when receiving requests on one interface and translate to a request for another interface. These patterns would be described on a high level so that they become independent of a communication interface.

These patterns could then be used in conjunction with specific interface-to-interface request mappings to create an interface controller in less time than implementing everything from scratch.

Examples of such patterns could be translation from asynchronous to synchronous communication or from synchronous to asynchronous. Another example of a pattern could be the behavior of sending a number of requests concurrently to each destination or to send each request sequentially.

1.3 IMPLEMENTATION METHODS

An important factor in software development is the time and effort it takes to develop a complete software system. In order to decrease the time and effort when implementing, a new generation of programming languages has started to appear. Those programming techniques are called fourth generation languages. Third generation languages are the most common languages today when implementing software. Languages such as Java and C++ are categorized as some of the most popular third generation languages.

Fourth generation languages are designed to allow the developer to automatically generate source code which is often a repeating task in third generation languages. The class and method structures and such are often automatically generated in these languages which mean that the developer can concentrate on writing the actual logic of the system.

The most commonly used programming languages today are also designed in a synchronous manner. That is the source code written is based on synchronous operations. For example a method call in Java or C++ from a point in the source code will wait until the execution of the method is complete and then continue to execute from the point where the method was called.

There are also languages which work in more asynchronous manners, although those languages are not as common as synchronous languages. An example of an asynchronous language is the Ericsson developed Erlang programming language. This language uses its own process model with lightweight processes which makes it easy to perform parallel tasks in the implementation.

Since the problems described in developing interface controllers often involve handling asynchronous interfaces, it could be implemented with less effort in an asynchronous based language.

There are also languages that are both asynchronous and fourth generation. An example of such as language is the MDA based Executable UML.

1.4 RESEARCH

There are two problems which are addressed in this report. Is it possible to create communication patterns which are common for interface controllers and could those patterns be implemented in a framework which can be used when developing new interface controllers. The second problem is the implementation method. Which kind of implementation method is the most suitable for developing interface controllers?

First a number of patterns describing common functionality used in existing interface controllers will be elicited. These patterns will then be used to design a framework which can be used to develop new interface controllers.

In order to find out how which implementation methods is the most suitable for developing interface controllers, the software will be implemented using four different implementation techniques.

The framework and an example interface controller will be implemented with a MDA language and three other different languages to compare them and see the different benefits of the languages to develop interface controllers.

The languages used will be Java, a third generation language, Rational XDE which is fourth generation development tool. Erlang will be used, which is a programming language designed to develop asynchronous systems. Finally Executable UML will be used, which is a language based on MDA. Executable UML is considered both fourth generation and asynchronous.

1.5 REPORT STRUCTURE

In chapter 2, *defining interface controller patterns*, we discuss and describe the problems and challenges that are in the scope of the problem domain. Concepts are introduced to set the scope for the whole problem description and to get a more detailed explanation of what issues needs to be solved.

Chapter 3, Limitations *and scope,* sets the boundaries for this master thesis coverage. The chapter discusses and describes the most important parts from the problem description and analysis. Things that are left out of the scope for this master thesis is important, but does not have to be included in the thesis to prove the point that are described in the *research methodology* chapter.

In chapter 4, *proposed solutions,* a conceptually solution is described. The solution describes different patterns that can be applied in conjunction with each other to form a solution. The solution described is not dependent or limited to any specific platform or technique.

In chapter 5, *implementation methods*, the four methods (Java, Erlang, XDE, xUML) in the comparison are explained and what characteristics they have are given.

In the *research methodology* (chapter 6) the methodology that is being used is described.

In chapter 7, *research results,* the results from the methodology are presented.

Issues that are not directly related to the research result will be discussed in chapter 8 (*discussion*). The writers' opinion on different matters, such as differences in the different languages and the tools used in the experiment, an evaluation of developing with Executable UML will also be included here. Future work will briefly be discussed here as well.

Chapter 9 which is the last chapter in this report summarizes and concludes what we have achieved with our work.

2 DEFINING INTERFACE CONTROLLER PATTERNS

The interface controller's task is to interface one or more different systems using those systems specific interfaces. These systems are called *providers* in the interface controller's context. The interface controller then provides a more convenient interface to a *consumer* system rather than the consumer communicating directly to the provider.

The consumer will send one or more request commands to the interface controller, which then translates the consumer requests into provider requests and sends them to the providers. Similarly the responses from the providers could then be translated back into responses to the consumer.

2.1 CONSUMERS

The consumer systems are systems, which are developed to use the incoming interface to the interface controller. The consumer interface is constructed in a way to make it more convenient for consumer systems to perform their tasks.

For example a consumer wants to communicate with providers, which use a synchronous interface for batch jobs, while the consumer wants to send its request asynchronously and poll the interface controller during the providers processes the requests. A consumer might also want to use a callback function, which the interface controller will call when the provider has finished the processing.

The behavior of multiple requests sent by the consumer can also be handled in various ways of the interface controller. For example if the interface controller maps a number of requests from a consumer into a single request to a provider, the interface controller could buffer the incoming requests from the consumer and when all requests for the mapping has been received, trigger the request to the provider. The interface controller could also verify the order in which the consumer sends its requests in order to decide which provider request to send and when to send it. There may exist solutions where many consumers want to communicate to the interface controller and the interface controller then communicates with one or several providers.

2.2 PROVIDERS

Providers are systems, which are already deployed and use some specific interface for communicating. The interface controller's job is to translate a provider's interface into another type of interface, such as a synchronous to asynchronous translation.

In an extended view of the interface controller it could also provide a single interface on top of several providers. In case there are two different scenarios of combining providers. In the first there are several providers of the different types, where requests are distributed among the providers by the type of request. In the second scenario, the providers are of the same type, where the requests are load balanced between the providers by using some balancing algorithm. Besides load balancing, sending different requests to several providers is also a possibility.

In the case of providing a single interface to many providers, the requests distributed among them, could be encapsulated in a transaction in order to rollback a request to a specific provider if a later request to another provider fails.

Also the behavior of sending requests to many providers should be able to be changed in the interface controller. For example, a behavior for a number of provider requests could be to send them concurrently to all providers at the same time, or to send them sequentially if a special order of requests is necessary.

2.3 REQUEST MAPPING

When translating the consumer's requests, the consumer sends a request to the interface controller using the consumer interface. The request could be translated into a different kind of request used when sending the request to a providers interface. A consumer's request could also trigger a number of new requests to a provider. The interface controller must have a mechanism for mapping consumer requests into another set of provider requests.

Also a consumer might send many requests, which maps into a single, or a new set of requests, for a provider. The different kinds of requests mapping can be seen in Figure 1.

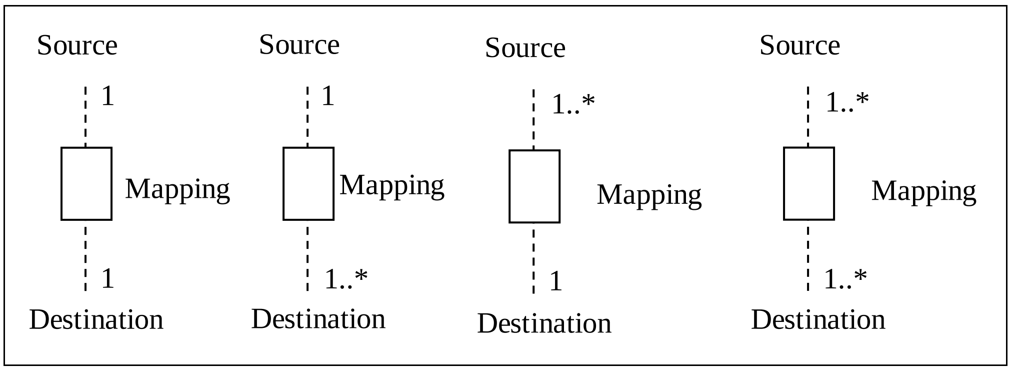


Figure 1: *Shows how requests can be mapped with different multiplicity.*

2.4 TRANSLATION PATTERNS

When developing interface controllers, there is much of the functionality involving translation in the communication that is implemented over and over again. For example the translation from an asynchronous interface to a synchronous interface is a recurring translation in interface controllers.

This recurring translation functionality can be described as patterns. When the patterns are created, a pattern can be applied to create components which can be reused in future development of interface controllers. The patterns are not specific for the given interface and can be used together in different combinations in different interface controllers.

The translation patterns we have found are described in the *solutions chapter*.

2.5 INTERFACE CONTROLLER DEPLOYMENT

The interface controller can be deployed in a number of different settings. In Figure 2 the most basic type is shown. The interface controller provides an interface to a consumer and translates it to a provider.

In Figure 3, the interface controller still provides the same interface to the consumer but uses two different kinds of providers. For example one request from the consumer can trigger two different requests to the two providers.

The most complex case, shown in Figure 5, the interface controller uses two groups of providers. The first group of providers is of the same type and could for example be used for load balancing requests of the same type. At the same time the interface controller can also use the other group of providers for other types of requests. This example could also be seen as the same in Figure 3 with an extra layer to the different providers to distribute requests among providers of the same type. In Figure 4 one consumer communicates with the interface controller and the interface controller then load balances between providers of the same type.

Besides load balancing, data spreading can be used. Data spreading can be used to send different requests to different providers instead of load balancing between the providers.

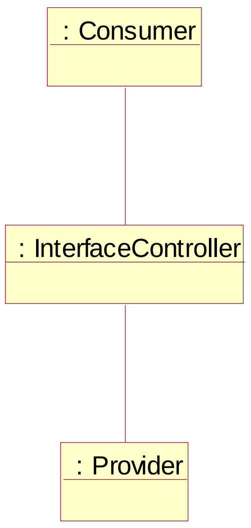


Figure 2: *Deployment of one consumer to one provider.*

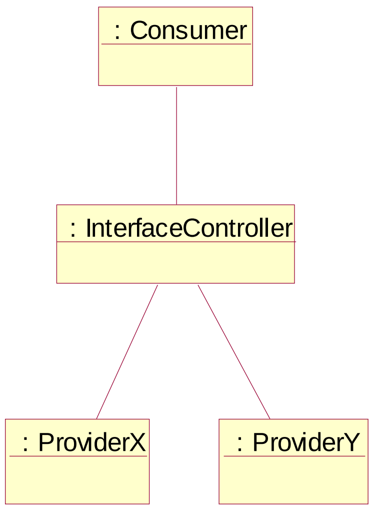


Figure 3: *Deployment of one consumer to different providers.*

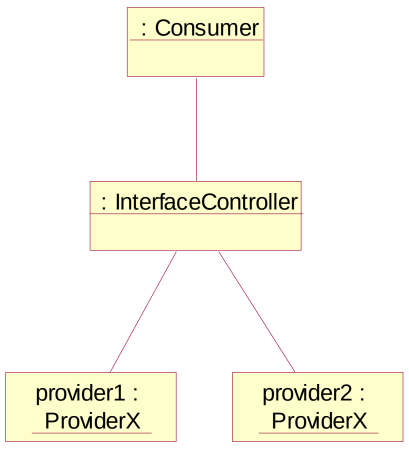


Figure 4: *Deployment of one consumer to many providers of the same type (load balancing or data spread over several providers).*

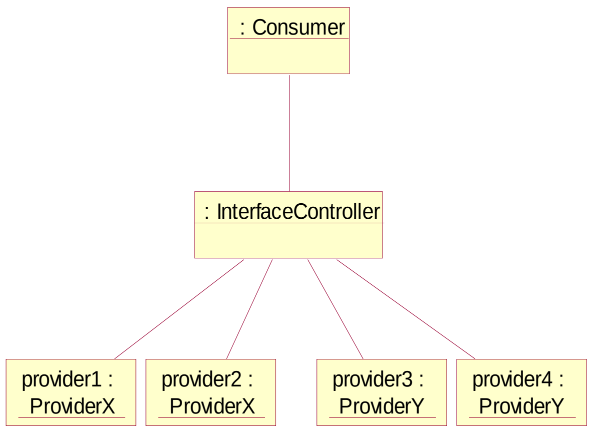


Figure 5: *Deployment of one consumer to different set of providers. Load balancing as well as data spread over several providers can be used.*

3 LIMITATIONS AND SCOPE

A large part of this project is to evaluate MDA and to see if Executable UML can be used to develop interface controllers and see if the development phase can be made easier and faster. Some complex parts are left out of the scope because of time restrictions for the project as well as complexity that are not needed to evaluate what we are looking for.

3.1 CONSUMER

When a consumer communicates asynchronously with the interface controller there are two ways for the consumer to see if the request is finished, and that is by using callback or polling. In the scope for this project polling is supported fully and callback will not be supported.

Consumers may use different protocols to communicate with the interface controller, such as SOAP and LDAP for example. In the interface controller those protocols are referred to as *Hosts*. In the scope for this project the interface controllers will support a *SimulatedHost*. The simulated host simulates the communication with external entities such as a consumer so that interface controllers may be tested internally to show that the model works.

3.2 PROVIDER

When communicating asynchronously to a provider it is possible to use callback or polling to see if the request is finished. Polling is in the scope for this project, callback is not.

The same way as the consumer in this project is being simulated, the providers will also be simulated, using a *SimulatedHost*.

3.3 REQUEST MAPPING

The different request mappings that exist is shown and described in Figure 1. All of the mappings except for the many to many mapping are in the scope of this project.

3.4 TRANSLATION PATTERNS

The patterns *timeout* and *retry* is within the scope of this project and will be partially supported. A request from a consumer may be divided into one or several requests to providers. Each of the requests from the interface controller to a provider can then either use the pattern timeout or retry. The limitation is that both patterns cannot be used for the same request from the interface controller to a provider.

Requests sent from a consumer to the interface controller can be sent using either synchronous or asynchronous communication; the same applies for the communication between the interface controller and the providers. Synchronous and asynchronous are core patterns in the interface controller and will be fully supported both from consumers to the interface controller and from the interface controller to providers.

The *transaction* pattern is within the scope of this project. Transactions will be used to bundle several requests from the interface controller to one or several providers.

The pattern *concurrency* and *sequential* are within the scope of this project and can be applied the same way as the transaction pattern.

3.5 DEPLOYMENT

The deployment shown in Figure 2 and Figure 3 are the two main deployments that we are looking into, both of them are in the scope of the project, and will be fully supported. The deployment in Figure 4 and Figure 5 shows how load balancing can be handled by using the interface controller to balance between several providers. Besides load balancing, data spread over several providers is also possible and is within the scope of this project. Load balancing involves several issues that make the interface controller much more complex to develop and configure. Session management has to be taken care of, as well as dynamically choosing which provider to send requests to. Load balancing is not in the scope for this project. The load balancing issue can however be solved on another level, which is handled by the consumer using the interface controller or some abstraction layer on top of the consumer.

In a deployment with one consumer to many providers of the same type is shown. This part is out of scope for this project. Load balancing might be done on a higher level as described earlier in this chapter.

Another method besides load balancing is data spreading as described in the analysis chapter. Data spreading, by sending different requests to providers will be supported.

3.6 EXECUTION

The interface controller framework created will be limited to only execute in model verifier directly. This limitation exists because if the framework is to be generated with the today's available model compilers, much hand written source code has to be made. In order to generate a complete system, the interfaces to the consumers and providers has to be written manually in the target language and then be combined with the generated source code from the framework models.

This limitation will not affect the testing of the patterns to see if they work and how they can be combined with each other.

4 PROPOSED SOLUTION

This chapter describes the proposed solution for implementing a framework for interface controllers. This solution is based on the analysis described earlier, but has been restricted to the limitations and scope also described earlier.

The solution described is on enough detailed level to describe the functionality of the interface controller framework, but not detailed enough to limit it to any specific development technique or platform.

4.1 TRANSLATION PATTERNS

In this chapter all of the found translation patterns are described. The patterns have partly been elicited from design documentation [3] and [4] for existing products, and from discussions with experts from Ericsson.

The patterns we describe in this chapter are not related to other existing patterns in literature. We define a translation pattern as: *A recurring solution to a communication problem in interface controllers.* Each translation pattern will have a description of the recurring *problem* it will solve, a *solution* that describes how the pattern will be implemented to solve the problem and finally *consequences,* which describes how this pattern affects the system when applied.

4.1.1 SYNCHRONOUS AND ASYNCHRONOUS

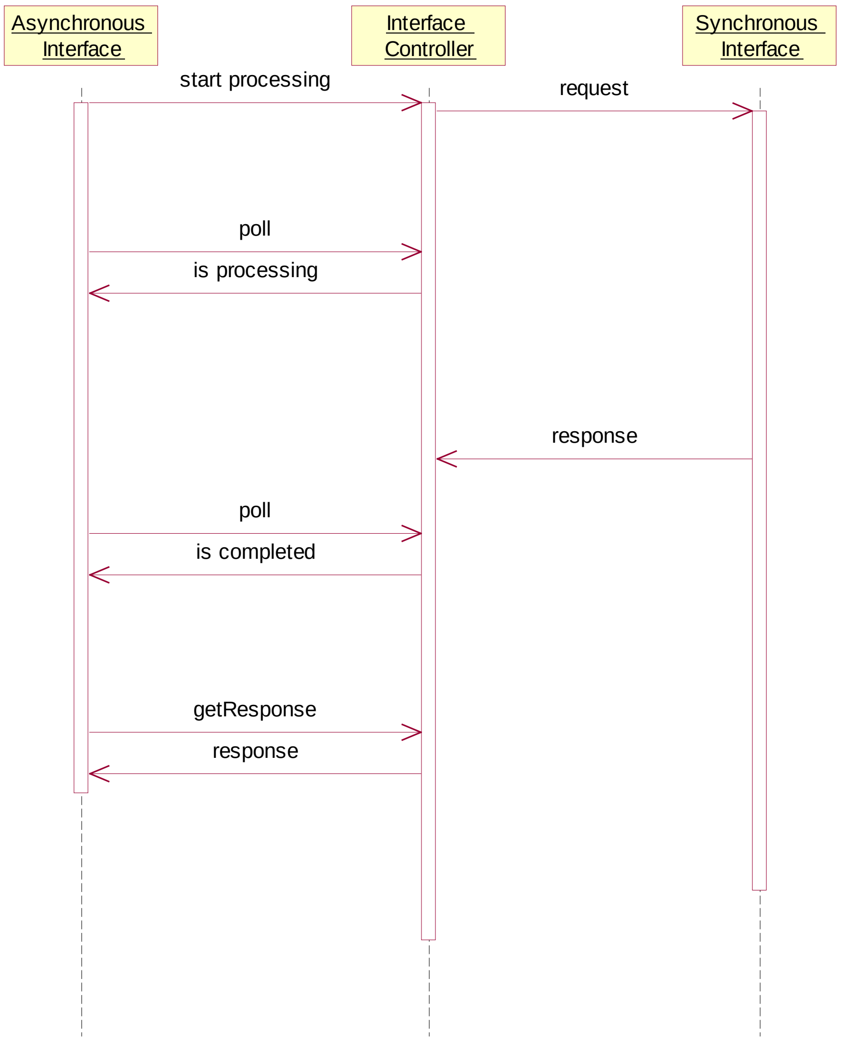


Figure 6: *Concept of using polling to see if a request has finished processing.*

4.1.1.1 PROBLEM

Synchronous communication is often used to process something, which takes some time to execute. To start the processing, a request is sent and when the processing is complete the request returns with a response. In the asynchronous case when a request is sent the sender does not have to wait for a response or the response is returned directly to the sender.

4.1.1.2 SOLUTION

When using the synchronous to asynchronous pattern asynchronous requests will be sent to the interface controller, which will map to a synchronous request. Asynchronous request can be sent to poll the interface controller to receive the status of the processing synchronous request on the other interface. An example of using polling is shown in Figure 6. Another option is to send a callback, which the interface controller will use to notify when processing is complete or if it has failed. An example of using callback is shown in Figure 7.

4.1.1.3 CONSEQUENCES

When using these patterns no special defined mapping for the incoming asynchronous requests can be defined, since the behavior of those requests are defined by the pattern itself. This pattern can only be used in an outgoing mapping of a single outgoing request.

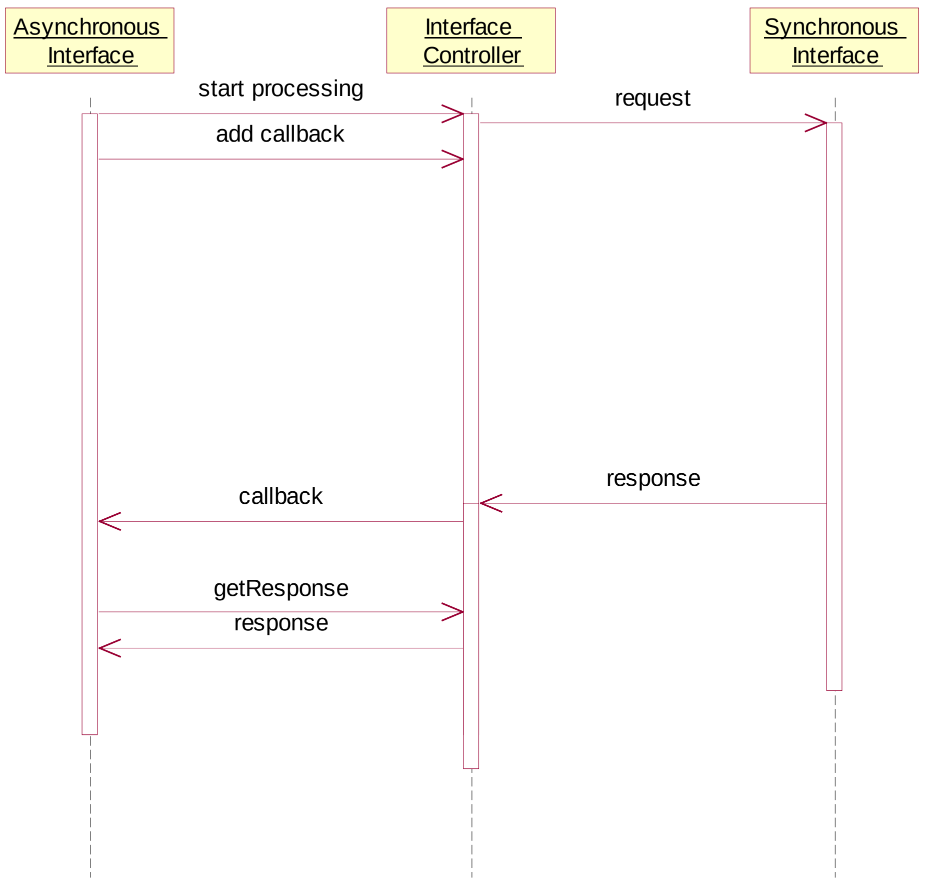


Figure 7: *Concept of using callback to be notified when a request has finished processing.*

4.1.2 Retry

4.1.2.1 Problem

If a synchronous request maps to sending several asynchronous requests and one of the asynchronous requests fails, it will fail the synchronous request as well.

4.1.2.2 Solution

A more efficient way is to retry the individual asynchronous request a specified number of times to ensure that the incoming synchronous request will succeed even if one of the asynchronous requests failed temporarily. The retry pattern is shown in Figure 8.

4.1.2.3 Consequences

A maximum number of retries should be specified in this pattern so the interface controller does not end up in a loop retrying forever.

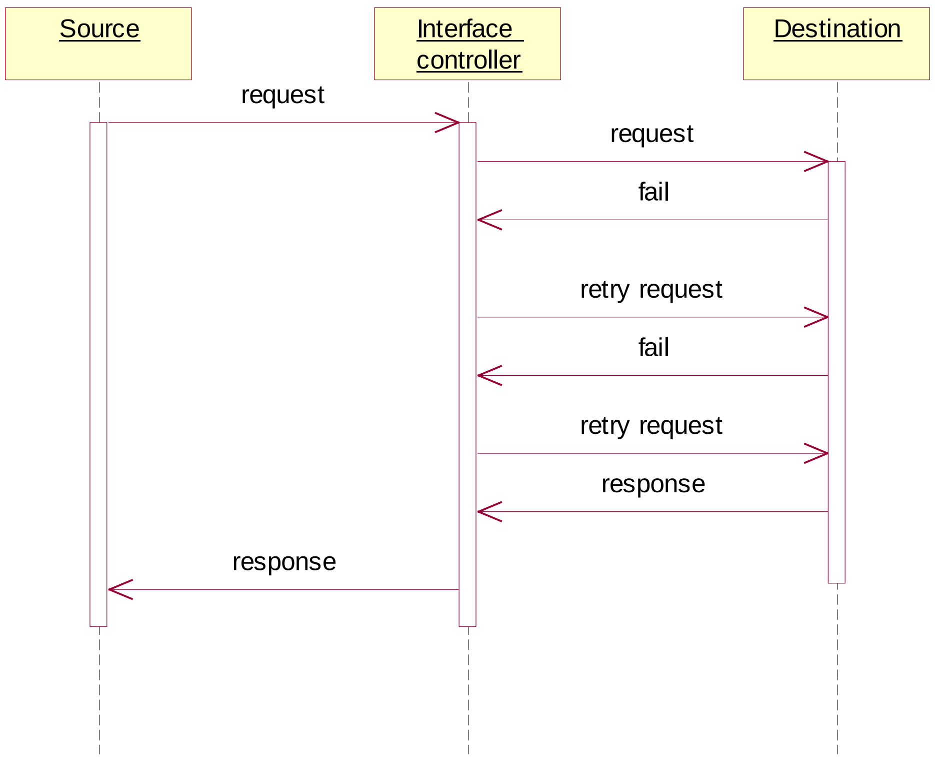


Figure 8: *Concept of using the retry pattern. A request fails and is retried.*

4.1.3 TIMEOUT

4.1.3.1 PROBLEM

If a synchronous request is sent to a destination, the interface controller will wait until the response is sent back from the destination. If the response is never sent back, the interface controller will get stuck waiting for the response.

4.1.3.2 SOLUTION

In this case it is useful to have a timeout for the request. If no response has been received after a specified amount of time, the request is considered a failure, and a failure response is sent back to the source. An example of a timed out request is shown in Figure 9.

4.1.3.3 CONSEQUENCES

If a timeout has occurred and a failure is sent back, the originating outgoing request could still be processed in the interface controller. This processing should be canceled or handled by the component implementing this pattern.

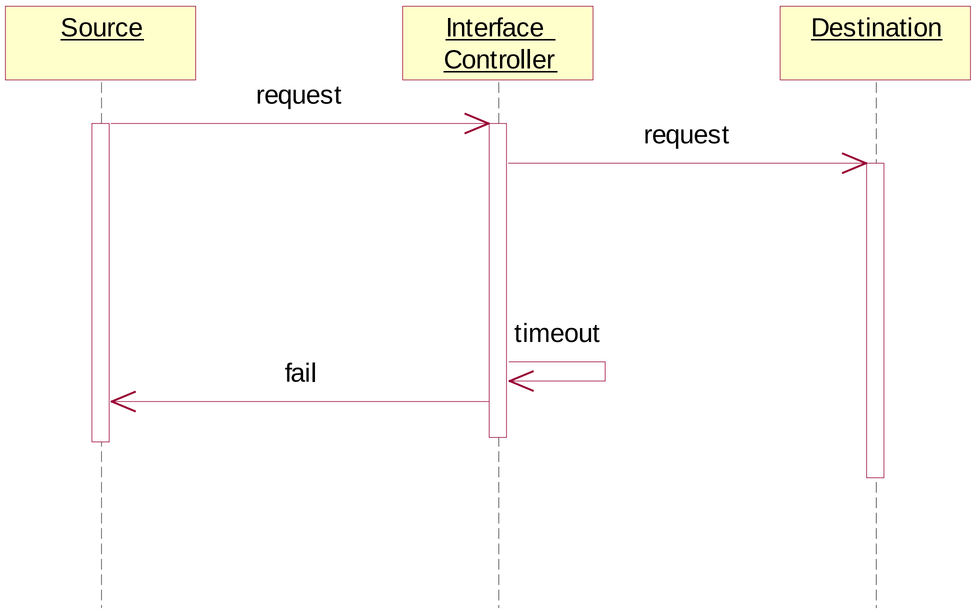


Figure 9: *Concept of using the timeout pattern. The request is not processed within the limit of the timeout and a failure message is sent back.*

4.1.4 SEQUENTIAL

4.1.4.1 PROBLEM

If a receiving end requires a specific order of its incoming requests, this order cannot be guaranteed if the requests are sent concurrently.

4.1.4.2 SOLUTION

Instead of sending the requests concurrently, the sending end will send one of the requests and then wait for the response and then send the next and so on in a sequential manner.

4.1.4.3 CONSEQUENCES

In order to apply this patterns the order of which the requests are required to be known of the sending end.

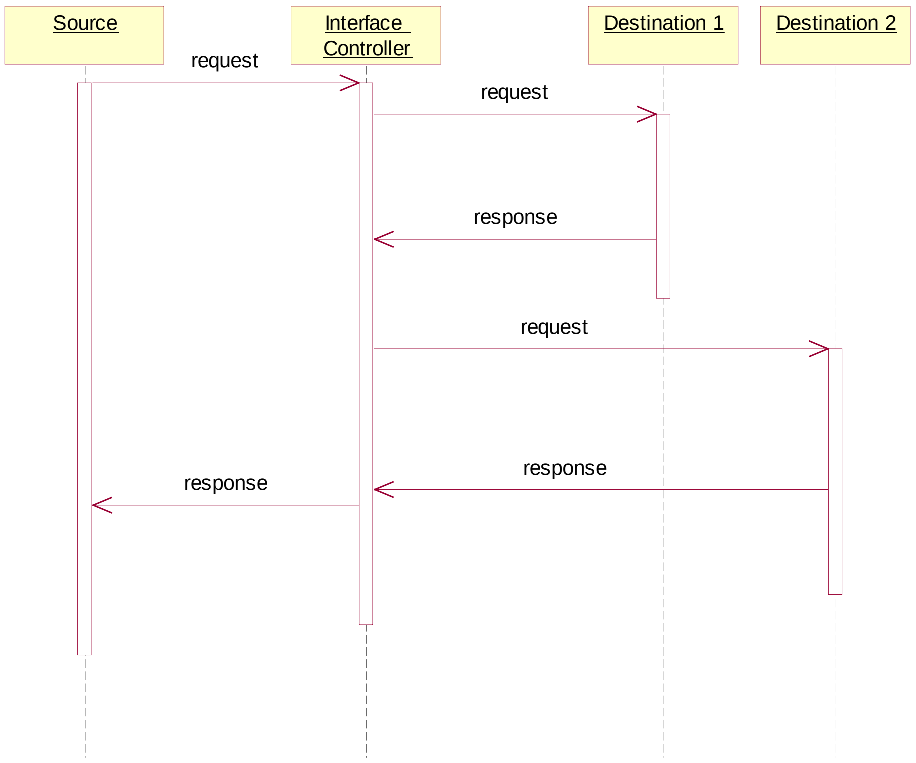


Figure 10: *Concept of the sequential pattern. Requests to destinations are sent sequentially.*

4.1.5 Concurrency

4.1.5.1 Problem

If a synchronous request consists of sending a large number of asynchronous requests to different hosts on a network, it could take a long time to send each request sequentially.

4.1.5.2 Solution

It could be more efficient to execute those requests concurrently instead of executing them sequentially. The interface controller will execute the requests in parallel and when all responses have been received, return the response to the synchronous request. This pattern is shown in Figure 11.

4.1.5.3 Consequences

If all outgoing requests are to be sent concurrently it must be ensured that all the receiving part can handle any order of incoming requests.

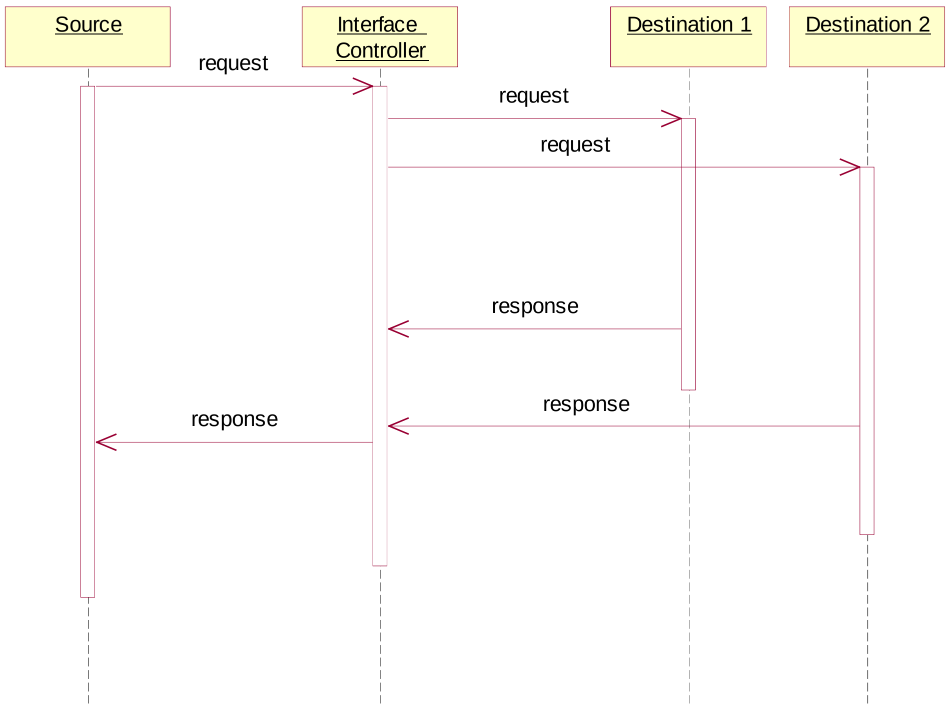


Figure 11: *Concept of the concurrency pattern. Requests to destinations are sent concurrently.*

4.1.6 TRANSACTION

4.1.6.1 PROBLEM

The communication between different nodes must be controlled in a way so that the state of the receiving end is always correct. For example if a synchronous requests maps into several asynchronous requests of which some succeeds and other fails, the incoming synchronous request also fails. But the mixture of successful and failed outgoing request could cause inconsistency in the receiving end.

4.1.6.2 SOLUTION

By setting up rules for different operations in a transaction-based manner, this can be controlled. An operation can for example be a synchronous to asynchronous translation, which involves several state changes such as several requests that has to be made for the operation to be successful. If one of the method calls fails the system must be set in a correct state again. A correct state can be assured either by doing a retry on the method call or a rollback is necessary to set the system in a correct state and report back that the operation was not successful. An example of a transaction is shown in Figure 12.

4.1.6.3 CONSEQUENCES

The receiving end of the transaction must support rollbacks to make it possible for the sending send to ensure the consistency of the receiving end.

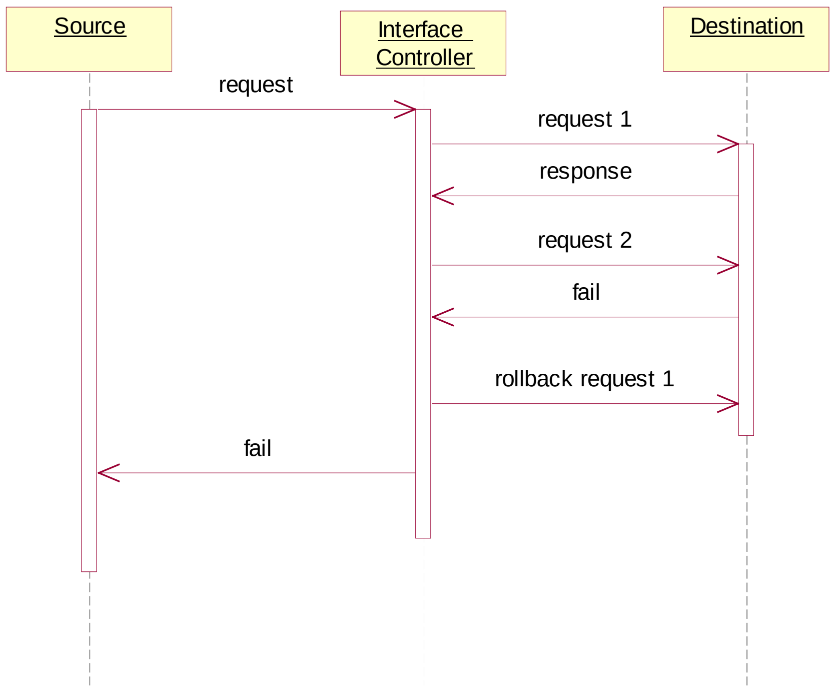


Figure 12: *Concept of the transaction pattern. When a request in the transaction fails, the transaction is being rolled back.*

4.2 RELATIONSHIPS OF THE INTERFACE PATTERNS

The class diagram in Figure 13 shows the concepts of the interface controller. The most important part is the *RequestProcessor* entity. This is an entity, which accepts a set of requests and forwards them to other *RequestProcessors*. The entire interface controller is built as a tree of *RequestProcessors*. An example of such a tree is shown in Figure 14. A subtype of a *RequestProcessor* is the *Host* entity, from which different kinds of hosts using a specific protocol are sending and receiving requests. The consumer is a host type of *RequestProcessor* sending its requests to the interface controller.

Next in the tree receiving the consumer requests are the translation entities. For example the translations between asynchronous and synchronous communication and the ability to encapsulate requests in a transaction are *RequestTranslators*.

Each of the translation entities has a *RequestMapping*, which defines the mapping of a number of requests into another set of requests. The *RequestMapping* has a *source* set of request, which then is mapped into a *destination* set of requests. For example the source requests can be a number of requests from the consumer, which maps to a single request to be sent to a provider. In this example, the interface controller could buffer the incoming requests from the consumer and when all the requests in the source mapping set has been received, the provider request will be sent.

A problem that could occur with buffering requests from the consumer is to know when a certain set of requests corresponds to a correct mapping to the provider. What if a subset of the buffered requests actually should be sent as a request to the provider itself? Issues like this should be brought some attention, although we will not focus on this issue in this project.

The different kinds of mapping of requests are shown in Figure 1 in chapter 2.

One set of requests has a defined behavior for sending its requests. This is depicted in the class diagram in Figure 13 as the *RequestSetBehaviour* class. The behaviors can for example be sequential or concurrent, meaning that if the sequential behavior is set for set of requests, each request in the set will be sent and waiting for a reply, then the next request will be sent and so on. In the concurrent case, all requests are sent concurrently.

Behaviors can also be set for each individual request. This is depicted in the class diagram in Figure 13 as the *RequestBehaviour* class. For example one specific request in a set can be retried a specified number of times if it fails or have a timeout.

Finally last in tree of the *RequestProcessors* (shown in Figure 14) are of the *Host* type again which corresponds to a destination provider.

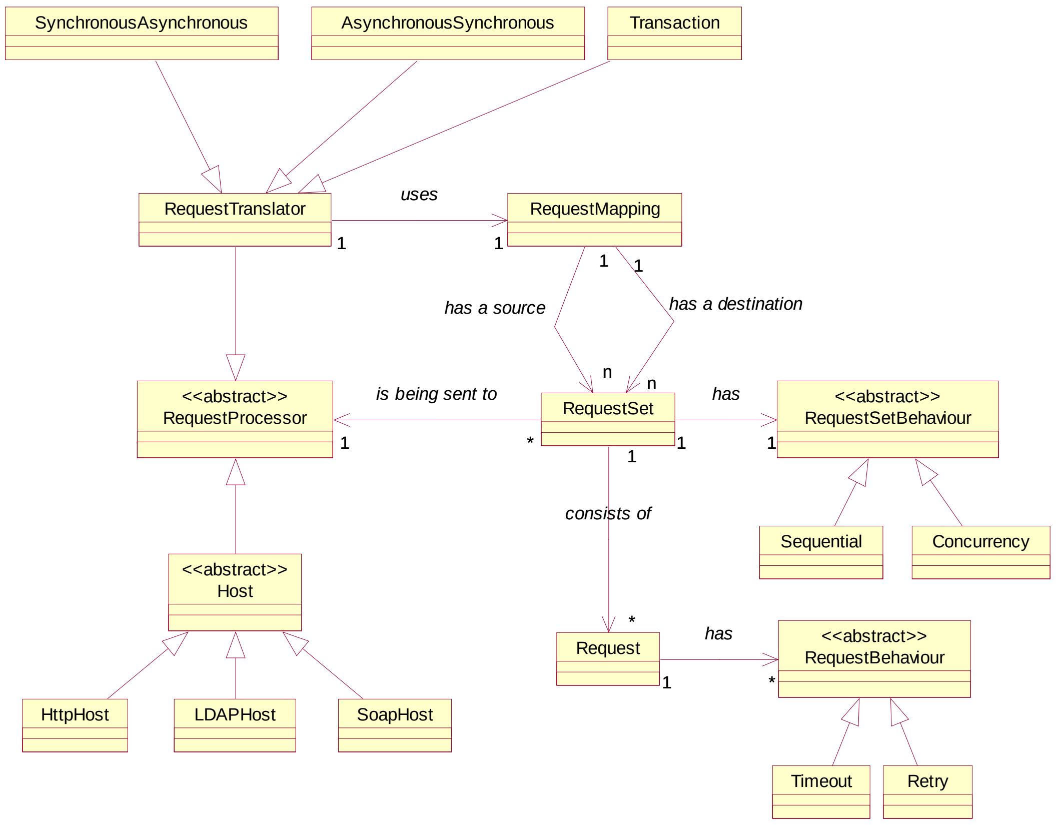


Figure 13: *The concepts of interface controller's structure are shown in a class diagram view.*

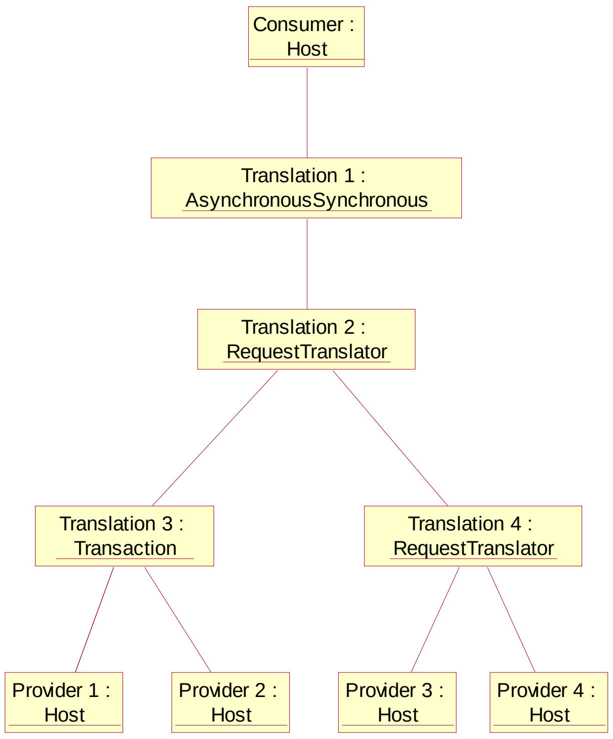

Consumer : Host
Translation 1 : AsynchronousSynchronous
Translation 2 : RequestTranslator
Translation 3 : Transaction
Translation 4 : RequestTranslator
Provider 1 : Host
Provider 2 : Host
Provider 3 : Host
Provider 4 : Host

Figure 14: *Example tree with RequestProcessors.*

4.3 REUSE

In order to effectively implement new interface controllers, most of the components described above could be reused from other interface controllers. The most important part in the reuse aspect is the *RequestTranslators*. Those entities are reused in new interface controllers in some combinations with each other to form the fundamental functionality of the interface controller.

The request mappings, which controls the mapping of data sent between the different *RequestTranslators* can be reused, but it is most likely that a new *RequestMapping* has to be created since those entities are probably different in each unique interface controller. The new mapping has to be modeled as a new Executable UML class in a modeler. In contrary to for example use XML files for defining the mapping, some form of logic is probably needed, which is hard to define without using some form of implementation language. Also if XML files are to be used, this means that hand written source code to parse the XML files has to be created directly in the target language.

For each request a *RequestBehaviour* can be reused or a new implemented. Similarly, the behaviors for the sets of requests can be reused.

The *Hosts* are also reused depending what interface protocols are being used towards the consumers and providers.

In order to create a new interface controller, some new entities has to be modeled. Those models create instances of the existing reusable classes in the framework. In addition also new classes can be created, which inherits from other existing classes in the framework.

5 IMPLEMENTATION METHODS

The different implementation methods and programming languages which will be used in the comparison (described in chapter 6) are introduced here to give more knowledge about them, and show how they differ.

MDA/xUML is the less well-known and the most different of the four implementation methods and therefore more attention is given to those parts in this chapter.

5.1 MDA

The software process has been developed a lot during the last 10 years. UML has become a standard for design and architecture that business people use to communicate and for programmers to use as a guide to develop correct software. The development of processes, how to go from design to finished software also changed a lot during the last years. Vendor tools are getting better and better.

An overview of the software development is described in Figure 15.

Still many programmers use only the code and some high-level architecture models as their only tool. The code then becomes the only way to see how the system really works and how it is really build. Today's 3G languages such as Java and C++ make it possible to get an overview of projects, although it's not always trivial, far from it (Figure 15 – 1).

Next step in development was to generate some kind of models from the code so that it was possible to visually view the system. The model is here a direct view from the existing code only.

A great improvement to the previous way of visualizing is to synchronize the code with the model as well as synchronizing the model with the code – so called roundtrip engineering (RTE, Figure 15 - 2).

At Figure 15 – 4 the models have such information that it was possible to generate a full implementation for a system. This way of using models is especially used in real-time embedded systems.

The last of the approaches described in Figure 15 - 5 are using models only approach. The source code should never be touched directly. [8]

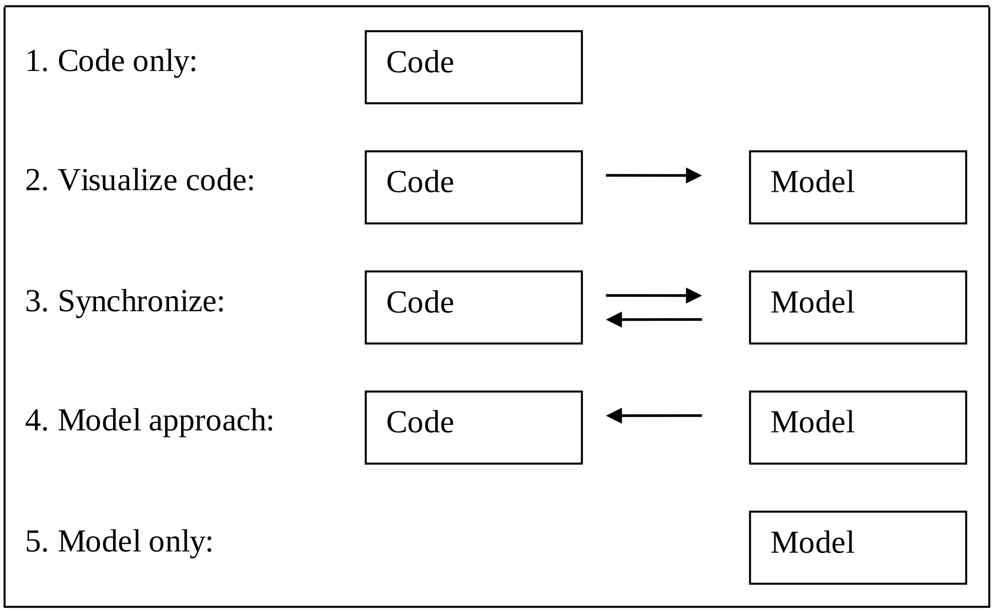


Figure 15: *Overview of how the connection between code and graphical models has evolved over time in software industry.*

MDA is short for Model Driven Architecture and is a framework for software development. It is about using a modeling language, as the actual programming language rather than using it like UML to design systems only. UML can be used to design systems and the output from UML is then being used for programmers to implement the system specified in the design and architecture. This mapping is time consuming and that is partially what MDA tries to solve by using models instead of manually transfer design to implementation. MDA has an approach where it is possible to divide business logic from its implementation fully. MDA is not dependent on any programming language, vendor or such things. The process of creating a MDA application can be described in some simple steps that will be explained in more detail later on in this chapter.

First thing is to create a Platform Independent Model (PIM). A PIM represents business functionality and can be modeled using UML syntax.

Next step is to create one or several Platform Specific Model (PSM). For example a J2EE/EJB model can be created. The model is partly generated from the PIM and partially written. When the PIM and PSM exist it is possible to generate source code for different platforms. This process is illustrated in Figure 16.

Some obvious benefits of this is easier integration in the future, different models can collaborate to reuse business logic for future MDA applications as well.

Object Management Group (OMG) [7] is an organization that works for standards and is currently working on the UML 2.0 standard that will soon be published. OMG also has specified important standards such as MOF that are cornerstones for the existence of MDA and its concepts. MOF will be described later on in this chapter. [2, chapter 3, 5] [9]

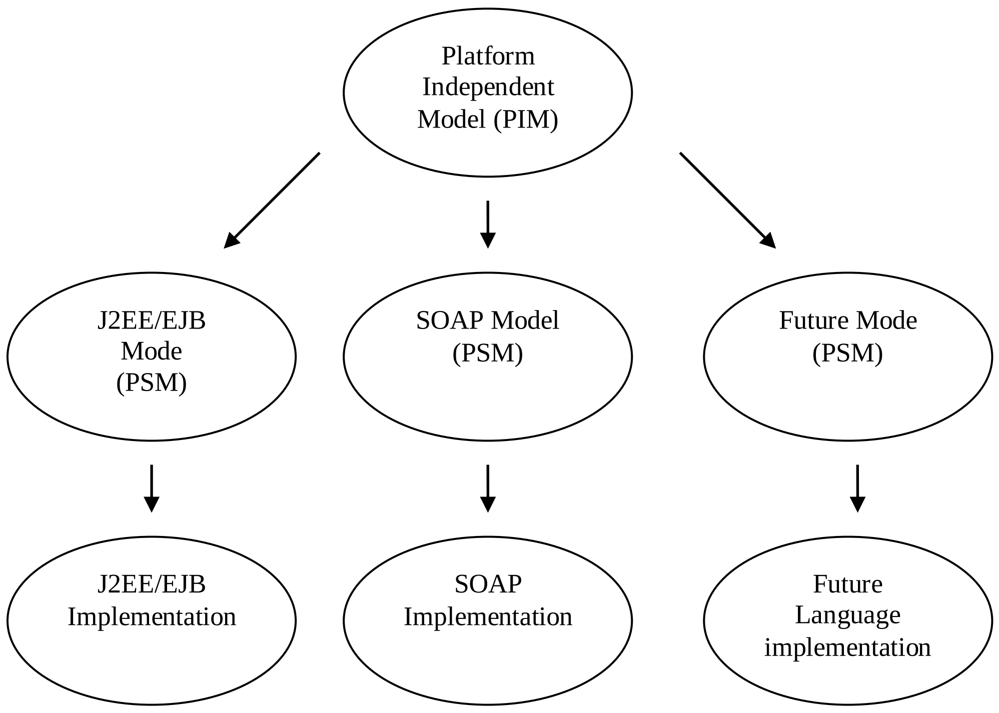


Figure 16: *Shows the process of transforming MDA models to ready to compile source code.*

5.1.1 MOF

MOF is a standard, developed by OMG, which is closely related to UML. MOF (metamodel) is a fundamental part of MDA and how models are described in a standardized way. It can be described as a language for describing other languages. For example UML can be described using MOF.

Historically compilers parse concrete syntax by creating a tree that reflects the specification. The trees can then be used to generate code. The same goes for metamodels; abstract syntax helps tools to construct abstract syntax trees that represent models.

The architecture of MOF consists of four different levels, called metalevels. [2, chapters 5, 8]

Level M3 is MOF. M3 elements are used to describe metamodels, more known as meta-metamodels. These elements are the core elements of metamodels. This level is used to define structure of a DTD for example. Level M3 elements are used to describe Level M3 elements.

Everything on level M2 is described using instances of M3 elements. For example are UML described at this level. The models at this level are called metamodels and by using XML DTD the structure of an XML document is defined.

The same way M2 models are populated by M3 elements, M1 models are populated by instances of M2 element. It is on this level that the usable models are being put together. The model layer represents a XML Document.

M0 elements are instances of M1 model elements. The user objects are here being modeled in XML.

5.1.2 ACTION SEMANTICS

OMG has defined a metamodel for an action language. The action language is part of UML and is supposed to work hand in hand with Object Constraint Language (OCL). OCL is a restricted rather static language where on the other hand action semantics is dynamic. The new standard creates a way for UML to describe dynamic actions completely so that it is possible to generate complete source code from the models. Action semantics has existed for a while and has been used mostly in real-time and embedded systems. Now when action semantics exists for UML it is more interesting for enterprise systems to start using it.

The standard for action semantics does not define the exact syntax to use for the different action statements. The standardization lies within a metamodel. So if different tools have different concrete syntax, they can exchange metamodels to be able to generate correct data. OMG has defined a metamodel for action semantics that many follow, for example the tool BridgePoint uses OMGs defined action semantics. [2, chapter 4]

5.1.3 PLATFORM INDEPENDENT MODELS (PIM)

First, to understand what a PIM is, the platform has to be defined, because that is what defines what the PIM is independent from. The term platform is a relative word; it can for example be used to describe operating systems as a platform. In this case a platform means middleware (technology). A PIM is not dependent on any technology such as .NET or J2EE/EJB. [2, chapter 8] [13]

5.1.4 PLATFORM SPECIFIC MODELS (PSM)

Just as the heading says, a PSM is a platform specific model, and with the definition of platform in the previous heading that means that a PSM is a representation for a specific technology, for example J2EE/EJB. The PSM then describes database models and models for the specific 3G programming language. A PSM is closely related to a programming language so to generate code from a PSM is not the hard part; the hard part is to transform a PIM to a PSM. These steps to transform between models and generate code are up to the available tools to implement in a suitable way. [2, chapter 8] [13]

5.2 EXECUTABLE UML

Executable UML is a concrete way to implement software, using MDA. It uses the technology and ideas described by MDA to actually develop a very detailed model using class diagrams and state chart diagrams. Executable UML defines a standard, which is stricter in the way you are actually modeling the software compared to MDA. In addition each state is also described using action semantics. The models are created in special software developing tools specialized for Executable UML. Those models can be compared to the platform independent model described in MDA. The platform specific model is then created by a model compiler, which automatically generates the complete source code and data of the software for a specific platform.

5.2.1 DOMAINS

A software system consists of a number of different domains, which together as a whole makes up the system. These domains can for example be the user interface, networking subsystem and so on. Each domain has its own clear mission statement or a specific task to handle. The different domains does not need to understand how the other domains work, but rely on other domains to perform services for the own domain. Domains can be elicited by creating use cases or analyzing the requirements for the system.

In Executable UML each domain is modeled as an executable model, which in detail specifies the behavior of the domain. The executable model uses other domains by having bridges to other executable models of other domains. The executable models and their bridges to each other then form the whole system.

Since the domains are modeled to implement a task in its own executable model, it can be easily replaced by another domain providing the same functionality as the previous one. For example, if there is a networking domain, which handles messaging over HTTP it can be replaced with another domain using a different protocol, but no changes are needed to be made in other domains. This of course also means that a domain created for this particular system can also be reused in development of another system.

Some domains of the system might already be implemented or a domain might also be an external system. These domains are called *realized domains* and do not need any modeling. Realized domains can also be used in Executable UML and are used and connected to each other just as any other domain.

Dividing the system into domains can also help managing the requirements of the system. Each requirement is assigned to a domain, which means a domain have a number of specific requirement, which in turn helps to better understand and model the domains.

The different domains will have dependencies on other domains and will thus transfer requirements from the domain to the domain it uses. Defining join points between two domains creates the bridge between them. The join points are defined correspondences between elements in the domains. [1, chapter 3]

5.2.2 CLASS DIAGRAMS

When the domains have been created, it shows the external view of the system and its requirements. In each domain the solution to these requirements must be modeled. The modeling is being done using class diagrams and attributes. The common characteristics and behavior of the entities, which constitutes the domain, are abstracted into classes.

The classes are created based on the conceptual entities of the domain such as things, roles, and interactions and so on. Each classed is filled with attributes, which captures the information and characteristics about the class. The types of the attributes may be domain-specific data types defined in the domain or core data types, which are defined by Executable UML.

In addition to the classes, associations between classes are created to show how the classes relate to each other. Each association has a specified multiplicity, which is later an important part when using the action language described later.

How to use relationships between classes are very strict in Executable UML and not all types of relationships defined UML may be used. For example aggregation and composition are never used. Generalization can be used, but the superclasses must always be abstract, which means object cannot be created that is only an instance of the superclass without being an instance of one subclass. [1, chapter 5, 6]

5.2.3 STATE MACHINES

To be able to generate a working system, the behavior of classes must somehow be specified. In Executable UML this is done using an action semantics language. In contrast to a conventional programming language, the action semantics language is designed to be used in conjunction with the class diagrams and its specified associations. The action semantics does not define any data structure or how data is stored, because this will be different depending on which target platform the system will be generated for. This is with other words handled by the model compiler. The action semantics are also designed to always be executed concurrently. An action semantics statement does not use any shared data, which could cause synchronization problems.

Actions are performed on instances of classes described in the class diagram. Those instances are selected from a given set of objects. Either one specific instance of a class can be selected or a multiple selection, which is held in a reference set, is selected. The instances in the reference set can be selected by different criteria using the **where** keyword.

In the first example in Figure 17 exactly one instance of class Book are selected. In the second example all instances of Book are selected into a reference set and in the third all books which are copyrighted 2002 are selected.

```
select any book from instances of Book;
select many books from instances of Book;
select many books from instances of Book where
        selected.copyright == 2002;
```

Figure 17: *Action semantics example of how to select one or several items.*

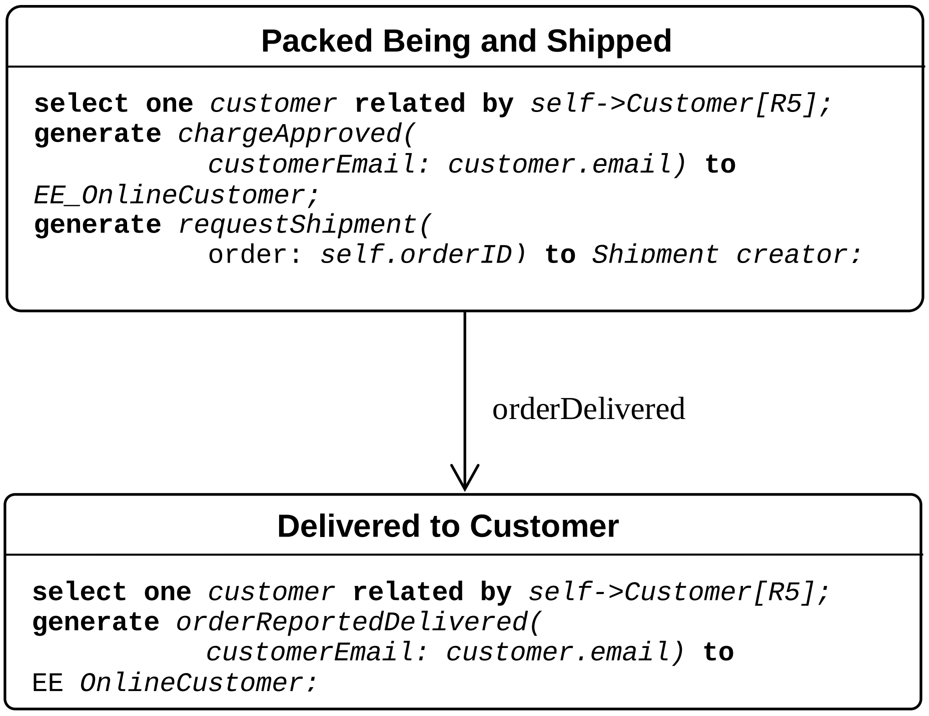
The action semantics can also select objects by using the relationships specified in the class diagrams, how this is done in described in the example in Figure 18. First the object *newPublisher* are related to a book object using the relation R1. The R1 relation is described in a class diagram. Then a select statement is used as described earlier but now selecting the objects using the specified relationship between classes.

```
relate newPublisher to newBook across R1;
select many allPublisherBooks related by newPublisher->Book[R1];
```

Figure 18: *Action semantics example selects items using relations.*

The action semantics are used in state machines. Each class will have a state machine, which consists of a number of states and the transitions to the different states.

An example of how the state machines looks like are depicted in Figure 19.



```
select one customer related by self->Customer[R5];
generate chargeApproved(
           customerEmail: customer.email) to
EE_OnlineCustomer;
generate requestShipment(
           order: self.orderID) to Shipment creator;
```

```
select one customer related by self->Customer[R5];
generate orderReportedDelivered(
           customerEmail: customer.email) to
EE OnlineCustomer;
```

Figure 19: *Example of two states in a state machine.*

In action semantics a signal can be generated to objects. Those signals are received as events in a class’s state machine, which will trigger a state transition into another state. The action semantics for this state will then be executed.

Figure 20 shows an example of generating a signal in the action semantic language. The signal itself can have a number of parameters. The parameter name to the left of the colon is the *formal parameter* and the name to the right is the *actual parameter*. Since the name of the parameters is paired with the value of the parameter, the order of the parameters does not matter.

The signal is received as an event, which triggers a state transition. In the new state the parameters of the event is read as shown in Figure 20. The *rcvd_evt* is a set of all parameters transferred from the signal.

```
generate addSelection(
           productID: rcvd_evt.productID,
           quantity: rcvd_evt.quantity) to order;
```

Figure 20: *Action semantics example of how to generate a signal.*

Another way to help describe the behaviour of classes in Executable UML is by the use of constraints. The constraints are specified for one or many attributes together in a special constraints language called the Object Constraints Language (OCL) and defines computed values and enables runtime checks.

These constraints could for example be used to make sure that some objects in the system are unique to other instances of the same class. The attribute in a class can be checked to be unique of all instances. In Figure 21 a constraint in OCL is created which does not allow an email address to be the same in different customers.

Another example of how to use constraints in OCL is depicted in Figure 22. Here we use a constraint define the value a derived attribute of a class. In this case the selection class holds a selection of item which each has a unit price. The derived attribute is defined to always be the total sum of all items. [1, chapter 8, 9, 10]

```
context Customer inv:
     Customer.allInstances()->forAll(p1, p2 |
                     p1 <> p2 implies
                                 p1.email <> p2.email )
```

Figure 21: *OCL example of how to restrict several customers to have the same email address.*

```
context Selection inv:
     Self.selectionValue =
          Self.quantity*Self.unitProceOfSelection;
```

Figure 22: *OCL example of a derived attribute.*

5.2.4 DOMAIN VERIFICATION

Since the models created in Executable UML are detailed enough to generate the source code for the complete system, the models themselves can be executed to verify the system just as it would have been done when testing source code. Dynamic verification of the models in Executable UML is possible by generating the source code automatically to the target platform by a model compiler or by using a kind of virtual machine, which interprets and executes the models directly.

Unit testing can be performed on a model by using scripts written in the action semantics language, and then test a specific class in an executable model.

Also, by combining several activities and initiate subsequent activities using action semantic scripts, system tests can be performed. [1, chapter 15]

5.2.5 JOINING DOMAINS

The system has been constructed by creating a number of domains which each constitutes of its own subsystem, but to make the system to work, there must be a mechanism to join the domains together to form the complete system. This mechanism is called *bridging* and in Executable UML there exist two ways to do this. The first is by using anonymous entities. Classes in one domain can send signals or invoke operations from this anonymous entity. The domain only needs to know the name of this entity, not in which domain it belongs. The external entities can also initiate signals into the domain, and thus works as a proxy between domains.

The second mechanism for bridging domains is to use *join points*. A join point is a correspondence between different domains. The correspondence can for example be a class-class join, where one class in one domain has another class in the other domain as a counterpart. The joins can also be between classes and instances of classes and also be correspondence between attributes and even behavior. This type of bridging is implicit and not maintained by the domain models. Exactly how the joining is maintained depends on the model compiler. [1, chapter 17]

5.2.6 MODEL COMPILERS

The model compiler compiles the Executable UML models into an implementation for a specific platform. For example a model compiler could generate source code in C++ for Solaris or another model compiler can generate the source code in C# for Windows. Any model compiler can be selected for a system developed using Executable UML and will generate the complete source code for the system. No additional programming in the target programming language will be needed. Also no changes or additions to the Executable UML models need to be made whether one model compiler is selected or the other.

The model compiler provides implementation to the different Executable UML elements. For example classes and attributes are elements of Executable UML, which the model compiler translates into C++ classes if the model compiler uses C++ or just some functions if it were to use C. The signals used in the action semantic language are another element, which could be implemented by using method calls. How the model compiler implements the elements is completely up to the model compiler itself.

The developer can control how the model compiler generates some specific parts of the source code by using the concept of *coloring*. When using coloring the developer gives input to the model compiler to use some specific way of implementing a part of the model. For example using coloring can control the type of an attribute in the target source code.

An important part of developing a system using Executable UML is to choose a suitable model compiler. Different model compilers are used for different kinds of systems. For example some model compilers favors performance and other may favor safety and use transactions for large parts of the generated system.

To choose the right model compiler, a third party compiler can be bought or a new model compiler can be developed. A special archetype language exists for defining how to implement the elements of Executable UML in a new model compiler. The model compiler itself could of course also be implemented totally in Executable UML itself. [1, chapter 18]

5.3 JAVA

The development of the programming language Java started out in the 1990 by a group of software engineers at Sun [12].

Java is a fully object oriented programming language. One of the goals with Java was to make it platform independent in the sense that compiled Java code should be able to run on different platforms. Compiled Java code is called byte code and that is the part that is platform independent, and can be moved between different platforms. Java programs are being executed using a virtual machine.

Memory management is easy to handle in Java compared to for example C++ where memory handling moves the responsibility to the programmers. Java uses a *garbage collector* to free memory, which takes the responsibility away from the programmer.

Java was developed to be used in a synchronous manner as most of the existing programming languages today. [11] [15]

5.4 RATIONAL XDE

Rational XDE is a next generation development tool based on the Java language. XDE is an entire integrated development environment (IDE) for java and adds new features to easier manage the source code being developed.

XDE supports both automatic code generation and reverse engineering of class and method structures. In addition XDE contains a large library of design patterns and which can be applied to classes to automatically generate source code from the patterns. The developer is allowed to define new design patterns and other constructs to later generate similar code to other parts of the system or to reuse them in other projects. [5]

5.5 ERLANG

Erlang is a general purpose functional programming language (developed by Ericsson AB) with many features such as concurrent processes, memory management and network distribution. The language was developed so that the lines of source code can be written compact.

When it comes to concurrency Erlang uses lightweight processes with dynamically memory handling. The processes have no shared memory and communicates asynchronous using message passing.

Like Java, Erlang runs in a virtual machine. An Erlang virtual machine can be seen as an Erlang node, where several nodes can exist in a network communicating. The different nodes can even be running different operating systems and Erlang will still work distributed.

Even though Erlang handles memory management using garbage collector it supports “soft” real-time support, meaning that the system requires response time in order of milliseconds.

If the performance in Erlang for some reason is not enough for some parts in an application the programming language C can be directly linked into it.

There exists a library called OTP which gives Erlang programmers access to a library with many useful functions such as IO, networking, math. [10] [14]

6 RESEARCH METHODOLOGY

The Model Driven Architecture (MDA) technique will be evaluated by applying the MDA techniques to a real industry problem, where the problem is that interface controllers today are implemented over and over again even though they use the same basic functionality.

A solution to this problem could be to discover a number of patterns describing the common functionality in existing interface controller implementations. The patterns have partly been elicited from design documentation [3] and [4] for existing products, and from discussions with experts from Ericsson.

The patterns we describe in this chapter are not related to other existing patterns in literature. We define a translation pattern as: *A recurring solution to a communication problem in interface controllers.* Each translation pattern will have a description of the recurring *problem* it will solve, a *solution* that describes how the pattern will be implemented to solve the problem and finally *consequences,* which describes how this pattern affects the system when applied.

Using the patterns, a comparison between 4 different programming languages/methods will be conducted. The languages/methods are chosen from 3rd generation languages as well as the 4th generation. From each of the two generations one synchronous and one asynchronous language are chosen. The languages are shown in Figure 23.

MDA will be represented by the Executable UML language, which is considered both a 4th generation language and asynchronous.

Java represents a third generation programming language and XDE represents a next generation language and Erlang an asynchronous language.

	3G	**4G**
Synchronous	Java	XDE
Asynchronous	Erlang	xUML

Figure 23: *The different languages/methods from the comparison.*

In the comparison effort and the performance will be measured. The performance will be measured in throughput in requests per second and the execution time in milliseconds to process one request. SLOC (source lines of code) is used as a measure of the effort required to implement the solutions. Besides the comparison, a discussion about the complexity of the different implementations will be made.

The definition of SLOC in our study is as follows. A physical source line of code is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character. Comment delimiters (characters other than newlines starting and ending a comment) were also considered comment characters. Data lines only including whitespace (for example, lines with only tabs and spaces in multiline strings) were not included.

All the tests will be performed using Windows XP running on an AMD Athlon XP 1700+ (1400MHz) with 512 Mb system memory. The java based implementations will be executed using Sun's Java Runtime Environment 1.4.2. The Erlang implementation will be compiled and executed using Erlang/OTP R9C. The implementation of the Executable UML implementation will be converted to C code by an experimental model compiler developed by Ericsson. The C code will be compiled using Microsoft Visual Studio 7 [6].

In the language comparison an example interface controller will be implemented in each language. In the four cases the interface controller described in Figure 14 in chapter 4.2 are being implemented.

The execution time test will be conducted by starting to measure time when the interface controller receives one request and until the interface controller sends back the response to originating request.

The throughput test will be conducted in a similarly way as the execution time test, with the exception that the time will be measured after a specific number of requests have been processed by the interface controller and then the time is divided by the number of requests sent.

In this comparison the measurements of the execution time and throughput are really the same, since the time to process one request in the throughput test is the same as in the execution time test. There could be other instances where the total time in a throughput test could differ. For example the times can vary because of how the implementation handles queuing of requests.

7 RESEARCH RESULTS

The first step was to create the recurring translation patterns found in interface controllers. This was done by examining design documentation of existing Ericsson products, which implements interface controller functionality.

The translation patterns found were:

- Asynchronous to synchronous interface translation
- Encapsulating requests in a transaction
- Concurrent sending of requests
- Sequential sending of requests
- Retry a request
- Timeout when sending a request

All those patterns are described in detail in chapter 4.1 on page 15. The next step was to create MDA models of the translation patterns. We choose to use the BridgePoint tool and Executable UML in order to create the models. This choice was made because Executable UML is one of the most complete implementation of MDA technology where the complete implementation of program logic can be modeled without writing any source code directly.

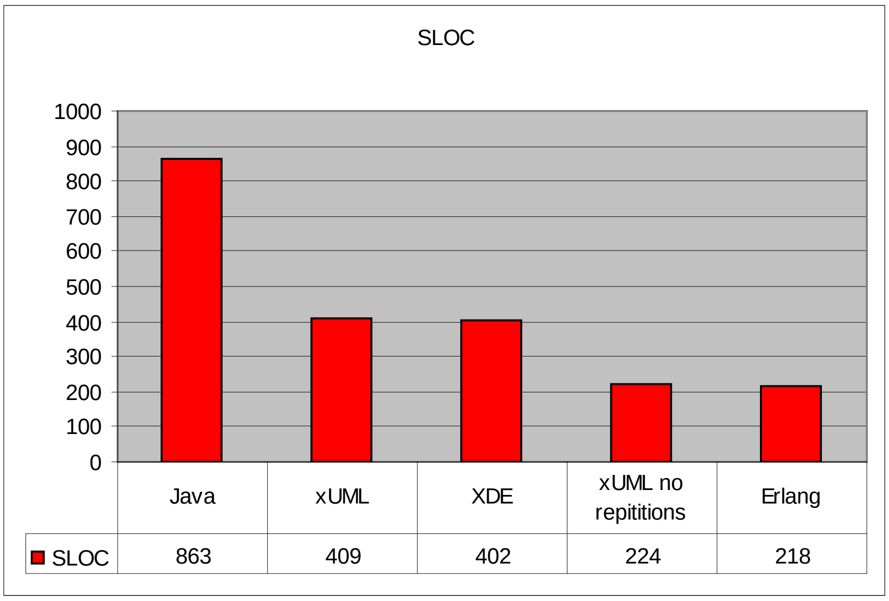
Each pattern was modeled as classes in Executable UML along other classes implementing a framework of reusable classes to develop new interface controllers. In the Executable UML model verifier, where the models are executed directly, each pattern could be verified for correctness. Also an example interface controller was created using other models to simulate the incoming and outgoing interfaces of the interface controller. This example was also executed in the model verifier to verify that the modeled pattern could work in combinations with each other.

The source code for the framework could also be generated by using a model compiler implemented at Ericsson. This model compiler allowed C source code for the Windows platform to be generated. This model compiler will only generate the source code for the logic of the system. This means that any particular interface, for example SOAP or LDAP, has be implemented manually in C and then bind this source to the existing models using *external entities* in Executable UML. Also any configuration file parsing and such has to be implemented the same way. Because of those limitations we did not implement a complete interface controller. Instead we used simulated interfaces and configuration modeled directly in Executable UML and verified the system by executing the models directly, since this was enough to verify and validate the results.

The second part of the result is the outcome of the comparison between the different languages shown in the above Figure 23.

In Figure 24 a graph over the amount of source lines of code that had to be written by hand is shown. The Java interface controller was written with 863 SLOC, in Executable UML the SLOC were 409, using XDE the lines of code were 402 and finally using Erlang the interface controller was written using only 218 lines of code.

Since the syntax of the Executable UML language differs much from the syntax of Java and Erlang. The language used in Executable UML has very few keywords and in order to perform a task, which in other common languages would just take up 1 line, would take up several lines in Executable UML. Because of this, we think that the comparison between Executable UML and the other languages are not entirely just. For this reason, we have also counted the number of constructs in Executable UML, which would normally be 1 lines of code in a traditional language.



	Java	xUML	XDE	xUML no repititions	Erlang
🟥 SLOC	863	409	402	224	218

Figure 24: *Number of source lines of code for each of the languages.*

The result shows that the fastest in the test is Erlang with 1250 requests/s.

Second fastest is Java/XDE with 323 requests/s. Slowest in this experiment is xUML with only 19 requests/s. The throughput with requests/s results are shown in Figure 25.

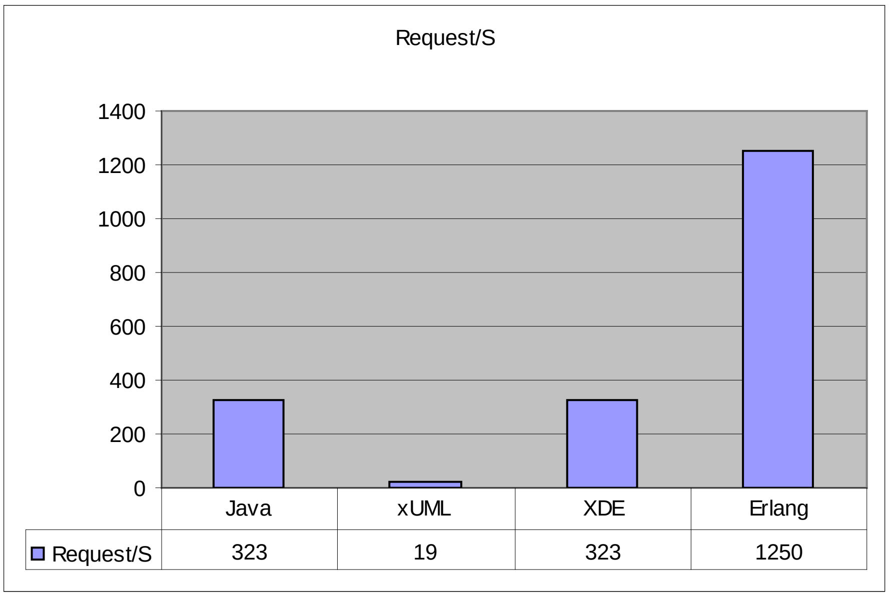


Figure 25: *Shows how many requests/s the different languages were able to send.*

Figure 26 shows a graph of how many milliseconds it took for a single request to be sent using the different languages.

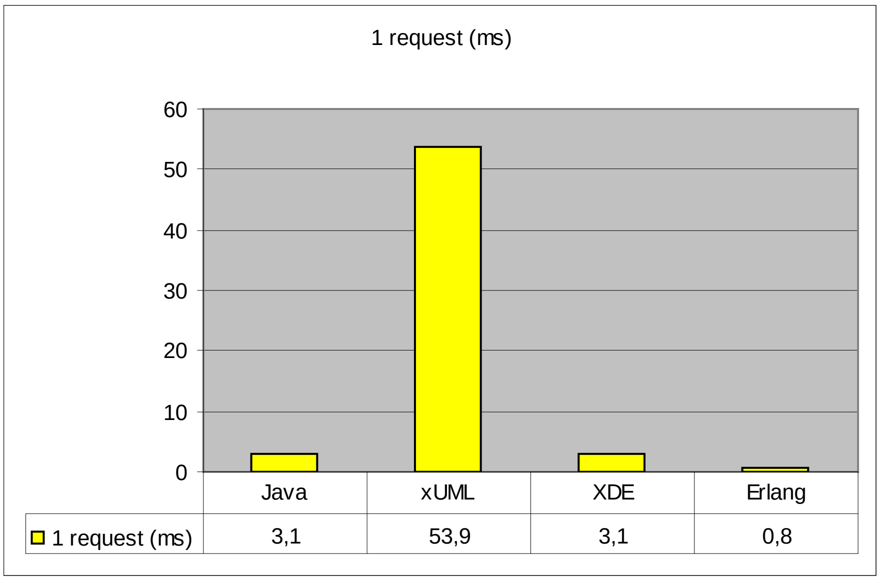


Figure 26: *Shows how many milliseconds a request took for each language.*

According to the results in the figures above, the fastest interface controller was written in Erlang. Also, when measuring the source lines of code Erlang was the one with fewest lines of code. The slowest of the implementations in the comparison were xUML with only 19 requests/s. When it comes to source lines of code, Java was the language that needed the most lines with 863 SLOC.

The execution time test were conducted by starting to measure time when the interface controller receives one request and until the interface controller sends back the response to originating request.

The throughput test were conducted in a similarly way as the execution time test, with the exception that the time was measured after specific number of requests had been processed by the interface controller and then the time were divided by the number of requests sent.

8 DISCUSSION

Discussion about issues, which are not results related to the research method

8.1 MDA

MDA is today not very widely used compared to traditional development. For MDA to be more established and more widely used, standards must be used so that tool vendors don’t have to use their own standards, which is not compatible with other vendor's tools. OMG and their partners are currently working on the UML 2.0 specification as well as the MOF 2.0 specification along with several other standards that are related to MDA. MOF is a framework for defining, manipulating and integrate metadata into platform independent models. When these standards are finished vendors can implement tools that support the standards so that data can be share between tools. When developing using model driven architecture today you are very dependent on the specific vendor you are currently using. Models developed in one tool are not compatible with other vendor tools.

8.2 EXECUTABLE UML

Executable UML is the most complete method of developing software based on the MDA technology in the sense of generating complete source code from platform independent models. It was for this reason we choose an Executable UML tool to develop our system.

Executable UML defines its own standard for defining the detailed models based on UML and the Action Semantic Language. A model compiler then generates the complete implementation as source code of the Executable UML models.

The advantages of using Executable UML are the enforced traceability between the software design and the implementation since in Executable UML the design is actually a part of the implementation. The design can be made in Executable UML as a simple class diagram and later developers can add more and more information to the model until it is detailed enough to be used to generate the source code. If an error in the design is discovered during the implementation and the developer fixes it by changing the implementation, the change will also be a change to the software’s design.

The biggest disadvantage of developing using Executable UML is that this is still a very new and maybe a somewhat immature technology. This is especially true for the model compilers. There exist only a very few model compilers today and most of them have many limitations of the control of the generated source code.

There also seems to exist a lot of misconceptions about what really can be achieved by using Executable UML. We think it is important to understand that Executable UML is really just another way of implementing software compared to the traditional way of writing source code directly in a third generation language. The procedure and problem solving of designing the software will still be the same and just as time consuming when using Executable UML.

Implementing a full-featured system will also be just as time consuming as if the system was written directly in a third generation language. This is simply because the implementation in the model will be at just the same level of detail when using a programming language. Also the Executable UML models can only define pure program logic. If the system is to use any platform specific interface, such as file access or network access using a specific protocol such as HTTP, the implementation handling those interfaces must still be provided outside of the Executable UML modeling and then be bound together with the Executable UML models.

Many seem to think that when using Executable UML, you implement a system by simply drawing some boxes and relations between them, and then execute the model compiler and you have a complete system. This is of course not the case at all. Executable UML is really just a new implementation *language*. It does not *solve problems* automatically.

8.3 THE BRIDGEPOINT TOOL

The most important advantage of the BridgePoint tool suite is that it is the most complete solution for developing using Executable UML. With this tool all models can be created in the modeler and the source code can be created using a model compiler.

The disadvantages of this tool compared to for example iUML, is that the different modeling views are presented in different windows. When modifying in one view it also often means you have to change something in another view too. When working with the modeler it is very easy to end up with a lot of windows of different view, which are quite hard to navigate between. This problem is better handled in iUML, where a tree view presents all different views of the models.

8.4 THE INTERFACE CONTROLLER FRAMEWORK

The problems with developing interface controllers can be partially solved by using a framework, where the common functionality of interface controllers is implemented. Instead of reprogramming the same functionality over and over again every time a new interface controller has to be created, the framework can be used.

All of the specified patterns were implemented in the framework. Also a test interface controller was created in order to verify that the framework actually could be used to create interface controllers.

From the start our intent was to build the framework so that it could be used to create a real deployable interface controller in a live environment. This however required that much handwritten C source code be to be created for the interface components. Instead our framework was only used to create testing interface controllers using simulated interfaces as Executable UML models.

The framework is a proof of concept of the framework and the patterns included in it really works. New patterns for other translation behaviors could be created and then tested in an interface controller created by using the framework.

The framework also shows that our architecture and design of the system works and there would be no problem to implement this design using a traditional way of development, such as Java for example.

8.5 LANGUAGE COMPARISON

By comparing different languages using only numbers wont give all the answers on which is the best language to use. If we would give some *expert comments* on the comparison between the languages and the complexity of them here is how we would order them by implementation complexity (most complex first):

- Erlang
- xUML
- Java
- XDE

There are other factors that should be taking in considerations when comparing the different programming languages, which is not showed in the measurements. The Erlang language scores very good in terms of code size and execution time, but some of the downsides of the Erlang language are that it is not very commonly used as for example java. This means if the Erlang language were to be used in a large corporation, a lot effort has to be made in terms like training personnel. Another downside of Erlang not being wide spread is that there is a lack of libraries to gain access to third party systems.

Executable UML suffers from the same disadvantages. Since Executable UML still is a very young technology it is not very wide spread either. There exists almost no "*realized domains*" to gain access to third party systems. Only pure application logic can be modelled in Executable UML.

As opposed to Erlang and Executable UML, the Java programming language is very wide spread and provides access through third party libraries to many other systems. Also many already trained java programmers exist.

A big advantage of Executable UML is the tight connection between the software design and the actual implementation. Executable UML also uses a GUI for modelling the entities and the very simple ASL language. This means Executable UML is much easier to learn compared to the other languages. Both of those advantages means that more people can be involved, for example designers or system architects, than when using the other common languages.

As shown in the language comparison, choosing a language with a higher abstraction level will decrease the effort required to implement an interface controller but will in one case have a significant impact on the execution performance. Choosing a language which is designed to be used for asynchronous systems will decrease the effort even more and also improve the execution performance.

The fact that the asynchronous language performs so well, we think is due to that the problems involved when creating interface controllers often are based on asynchronous communication.

It should also be pointed out that since we were conducting the implementation of the interface controllers ourselves, there is a risk of the comparison being unfair since we are more experienced and knowledgeable with some of the implementation methods than the others.

8.6 FUTURE WORK

In order to make the Executable UML models of the interface controller framework really usable, the code for handling the interfaces has to be developed. Although there is much work to be done, the different interfaces could be developed in C and then be bound to the already completed models. This way the framework could be used to continue develop the framework to eventually be used to create interface controllers which can be used in real life.

9 CONCLUSIONS

Reusability in interface controller development can be achieved by creating implementations of reusable communication patterns. The communication patterns described in chapter 4.1 can be reused for developing new interface controllers.

When using a fourth generation language less effort has to be made when developing an interface controller compared to the Java language, which represents the current development method. In one case the performance will worse compared to the third generation languages.

When using an asynchronous language the effort will be cut down even more than a fourth generation language, but the performance will also be better than compared to a synchronous language.

In the language comparisons it is shown that the resulting system developed using Executable UML will be much slower in execution time compared to the other languages. Using a fourth generation development tool like XDE and Java, the resulting system will execute faster but were written with a higher source code size as Executable UML. The same system could be developed in the asynchronous Erlang language, which would have faster execution time and written with less amount of source code than all the other languages.

When looking at the measurements between the languages it clearly shows that an asynchronous language would be the best choice for developing interface controllers.

REFERENCES

[1] Executable UML: A Foundation for Model-Driven Architecture, Mellor Stephen J., Balcer Marc J., Addison-Wesley 2002

[2] Model Driven Architecture: Applying MDA to Enterprise Computing, Frankel David S, OMG Press 2003

[3] CPM Software Architecture Description, Ericsson

[4] SOG System Description, Ericsson

[5] IBM, http://www-130.ibm.com/developerworks/rational/products/xde, last visited 2004-10-20.

[6] Microsoft, http://msdn.microsoft.com/vstudio/productinfo/default.aspx, last visited 2004-10-20.

[7] Object Management Group (OMG), www.omg.org, last visited 2004-10-20.

[8] An introduction to Model Driven Architecture - Part I: MDA and today's systems, Brown Alan, IBM developerWorks 2004

[9] MDA: Revenge of the Modelers or UML Utopia?, Thomas Dave, 2004

[10] Erlang, http://www.erlang.org, last visited 2004-10-20.

[11] Java, http://java.sun.com, last visited 2004-10-20.

[12] Sun Microsystems, http://www.sun.com, last visited 2004-10-20.

[13] MDA in Enterprise Architecture? The Living System Theory to the Rescue, Wegmann A. Preiss O, 2003

[14] "Erlang", Heller Martin, journal: byte.com, 2003

[15] Jace: A Java Environment for Distributed Asynchronous Iterative Computations, Bahi J. Domas S, 2004