Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

and

University of Twente - The Netherlands

August 2003

Compose *

A Runtime for the .Net Platform

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Carlos Francisco Noguera García

Promotor: Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Lodewijk Bergmans (University of Twente)

Contents

Chapter 1

Introduction

1.1 Motivation

It is a common belief that the research and industrial branches of computer science are two different and well delimited areas. Usually advances on research take a considerable amount of time before they are adopted by the industrial side; take for example the Object Oriented programming paradigm, which is just now, with the advent of Java, starting to permeate the industry, whereas it has been around the research community for over 20 years. This happens for a variety of reasons, most of them out of the realm of computer science and well into politics marketing, economics, etc. But also because of the fact that some of these advances are difficult to incorporate into existing solutions.

However, in recent years there has been a renewed interest in accelerating the time that takes for academic research to reach industrial approach; this is the case for the development of tools, such as AspectJ, that introduce cutting edge concepts -Aspect Oriented Programming- on a main stream programming language, Java. This is just one of the multiple projects that start from the fact that it is not enough for a concept to be good idea, it must also be easy to incorporate into existing solutions; another project is the Composition Filters Model.

It is the purpose of this thesis to exploit one of the attributes of Composition Filters, its language independence, by integrating it with the .Net platform.

1.2 Contents of this Thesis

This thesis is organized as follows. In chapter 2 some background and the problem statement (2.4) are introduced. A small summary about Aspect Oriented Programming (2.1), an overview of the Composition Filters Model and the previous its implementations (2.2),and some work that is related to this one (2.5) make part of this background.

In chapter 3 the Compose* System is presented. The requirements of the project (3.1), its design (3.2), and implementation (3.3). Finally, in chapter 4, the product of this project is evaluated (4.1), some concluding remarks (4.2) and proposals for future work (4.3).

Chapter 2

Background and Problem Statement

2.1 Aspect Oriented Programming

Whenever making changes to any non-trivial application, programmers are faced with a long and error prone process: first they must find the place in which changes must be applied within a large code base; with some luck, the number of affected modules is small, which means that the amount of code to understand is equally reduced. However, these modules may not contain only the code they are interested in, that is, the module might contain several *concerns*; adding the difficulty of understanding and adapting artifacts that have nothing to do with the original problem to an already difficult task.

These mixed modules are a common problem in any program, and call for a clear *separation of concerns* or "The ability to identify, encapsulate and manipulate only parts of software that are relevant to a particular concept, goal, or purpose"[TO01].

One of the techniques for separating concerns is known as Aspect Oriented Programming (AOP). It deals with *crosscutting concerns* by defining aspects as modular units of crosscutting implementations. This permits the programmer to place the intersecting concern on a different constructs in a reusable manner.

It is important to note that Aspect Oriented Programming is not only a technique to that can be used to modify existing applications. Currently research is advancing on development methodologies that include AOP from the start, for example, see [KJS00]

Various implementations dealing with AOP exist, in order to take advantage of the current research, some of the most relevant to the Compose* system are treated in 2.5.

2.2 Composition Filters

The Composition Filters Model (CF) is an extension to the Object Oriented model. It is closely related to Aspect Oriented Programming, in the sense that with CF it is possible to model Aspects, but CF's origins date farther back in time. The base concept in CF is that messages that enter and exit an object can be intercepted, and manipulated in various forms modifying the way in which the object behaves. Also, in CF it is possible to state on which objects will the filters be placed, and how groups of modules will interact with each other. Both these characteristics will be discussed in more details in the following sections. For a more in depth look at the CF model, see [BA]

2.2.1 Filters

Filters are the atomic unit of the CF model. There are various types of filters, all sharing a common structure:

- A name that identifies the filter,
- The type of the filter and
- A set of expressions that define the way that messages are to be filtered.

Filters can refer to internal and external objects, as well as to the two types of methods defined in the model: *regular methods* and *condition methods*, referred to as conditions. Regular methods define the behavior of the object, and conditions implement side-effect free boolean expressions about the state of the object.

Filters are defined as part of the visible interface of the object in which they are superimposed. This interface declares the objects that the filters refer to as *internals* or *externals* and methods are declared as regular methods or conditions. In this interface definition, filters are grouped into filter modules. A simplified representation of the CF model can be seen in figure 2.1.

The filtering process is described in the figure 2.2. In it, arriving messages are, one at a time, passed through each of the *filters*. In each filter, messages are matched against *filter patterns*; if the message does not correspond with any of the patterns on a filter, it passes to the next filter. If there are no more filters left, an error is produced. Depending on the filter type messages are transformed when they match or not.

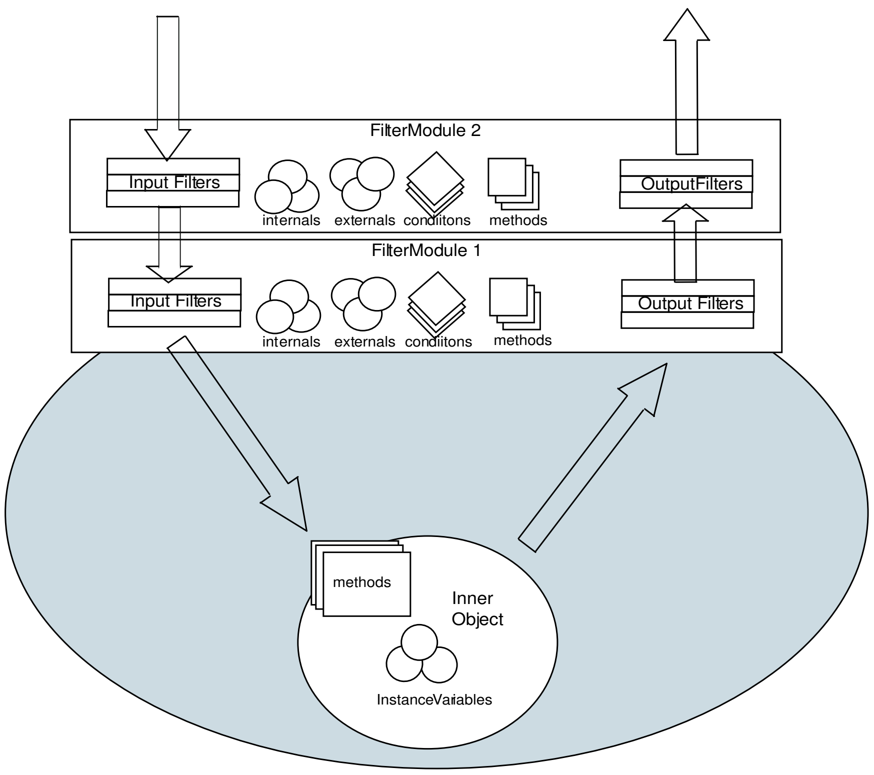

Figure 2.1: Representation of the CF model

Filter Types

There are various filter types whose behavior is defined by what actions are taken when it accepts, that is a message matches any of the patterns defined for the filter; or rejects a message. The most common filter types are dispatch, error, meta, send, and substitution. Their behavior is explained in table 2.1

2.2.2 Previous implementations of Composition Filters

There have been various implementations for the CF model. Dating back to 1995, the CF model has been in various platforms (smalltalk, C++, Java) and in various manners (compile-time and runtime). In the next sections there is a small overview and their relevance to the Compose*system.

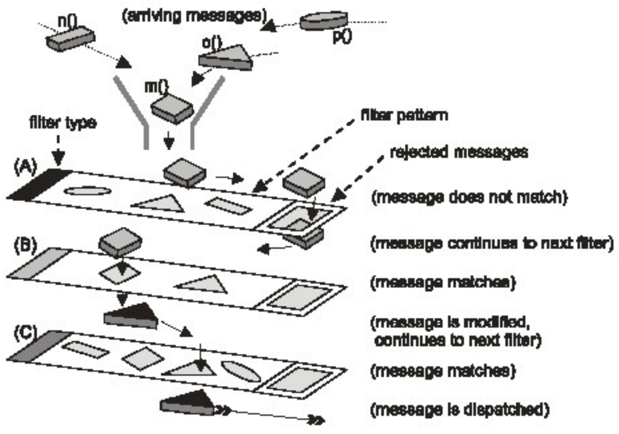

Figure 2.2: Graphical representation of the filtering process

Sina/ST

This an implementation in Smalltalk of the Sina language which incorporates the composition filters object model. Sina/ST includes a programing environment, a compiler and a kernel that provides runtime support. For a more detailed look at Sina/ST see Piet Koopmans master thesis[Koo95]

Sina/ST uses smalltalk's *message not understood* mechanism to intercept messages sent between Sina/ST objects, and presents them to the language's kernel. This kernel performs the filtering process and takes the necessary action. This kernel supports concurrent messages, input and output filters and five filter types: Dispatch, Error, Send, Meta and Wait.

Sina/ST is relevant to the Compose* project because it follows the same approach of a runtime environment, and therefore, Compose* and Sina/ST are every similar in design. However, since Sina/ST is a language by itself and not an extension to the smalltalk language, it prevents the interaction between smalltalk objects and Sina objects; this contradicts one of the goals of the Compose*system: the interoperability between native .net objects and Compose*objects.

Filter Type	**Accept Action**	**Reject Action**
Dispatch	The message is redirected to another object defined in the filter specification.	The message continues to the next filter.
Error	The message continues to the next filter.	An exception is raised stating the error.
Substitution	The message is substituted as directed by the parameters of the filter specification.	The message continues to the next filter
Meta	The message is reified, and delegated to another object.	The message continues to the next filter.
Wait	The message continues to the next filter.	The message is blocked until the internal state of the object allows the filter to accept the message. During the time when the message is blocked, the object is free to receive other incoming messages.

Table 2.1: Definition of the Filter Types

ComposeJ

ComposeJ is an implementation of the CF model on top of Java. It is constructed as a preprocessor to the Java compiler. It includes a parser for the specification of the composition filters, and said preprocessor. Since it is a compile time tool it does not need a runtime kernel. In the current ComposeJ implementation, two filter types are implemented: Dispatch and Error. For a more in depth look at ComposeJ see J.C. Wichman's masters thesis [Wic99].

ComposeJ bases its implementation of the CF model on successive source code transformations directed by the composition filter specification. The concepts of CF are then translated into Java statements. This takes away the dynamic nature of Sina/ST, but it allows for ComposeJ objects to interact with regular Java objects since there is no difference between them.

ComposeJ is relevant to the Compose* system because it supports the interesting feature of interaction between CF objects and platform objects. Also it tests the

CF model against a strongly typed language such as Java, which presents issues that must also be dealt with in Compose*. However, ComposeJ being a compile time approach to CF, it is implemented in a very different way than how a runtime system would be implemented; so lessons learned in it are not translated into Compose* in a straightforward manner.

ConcernJ

ConcernJ implements concepts of the CF model that are related to superimposition as a preprocessor to ComposeJ. Because of this, it is similar to ComposeJ. ConcernJ uses concepts from OCL and UML to define where and how should filters be placed on top of objects. For more information, see Patricio Salinas' master thesis [Car01]

ConcernJ does not have much relevance with the current state of Compose* because no development for superimposition is required; however, as the system evolves, the need for a clear method for filter superimposition will be present; and the system must take into account the needs of this in its current design.

2.3 The .Net platform and J♯

During the October 2000 Professional Programmers Conference Microsoft launched its new initiative .Net. Geared towards enabling software as a service, .Net integrates a number of technologies that emerged from Microsoft during the late 1990's as well as new additions: COM+ components, ASP web development, XML, support for new protocols like SOAP and UDDI; and a general focus on Internet. The .Net platform consists of four product groups: development tools, specialized servers, web services and devices.

The most important component of the .Net Framework is the Common Language Runtime (CLR). It is in essence a virtual machine in charge of activating objects, perform security checks on them, lay them out in memory, execute them and garbage collect them. The CLR takes these objects from Portable Executable (PE) files, which contain the .Net equivalent of bytecode, called Intermediate Language.

The CLR provides one of the features of .Net that is interesting for Compose*: language interoperability. This feature is derived from the fact that all languages on the .Net platform (Visual Basic, C++, C♯, J♯, etc) are compiled to IL and executed that way. This means that objects programmed in different languages can communicate without problem since their differences disappeared when they were compiled to a common language, IL.

2.4 Problem Statement

Design on the Object Oriented paradigm lacks the means for modelling certain aspects that often present themselves during development, as well as for accurately representing modular abstractions, these deficiencies are identified in [KLM+97]. It is the purpose of this work to enrich the .Net platform by adding CF concepts that address these issues.

More specifically, the extension to the .Net platform will be composed of a runtime system for the interpretation of CF specifications. This runtime must be of use when developing completely new applications, as well as when working with legacy systems.

2.5 Related Work

Aspect Oriented Programming is a popular new topic with many projects currently developing. Due to AOP's closeness to CF, most of this projects are related to Compose*. Below there is a small overview of some of these relevant to Compose*. AspectJ 2.5.1 is interesting because of its impact on AOP, as well as its .Net port AspectC♯ 2.5.3. EAOP is a different approach to AOP, and it is interesting because of the dynamic nature of its approach.

2.5.1 AspectJ

AspectJ[KHH+01] is an aspect oriented extension to the Java programming language. It is arguably the most popular approach to AOP, and is geared towards industrial software development.

AspectJ adds the concept of *joint-point* to Java , that is a well defined point in the program execution, and *advice*, that is the code that is executed once a joint-point is reached. In addition to these, AspectJ introduces *inter-type definitions*, which allow the programmer to modify the static structure of classes (methods and relationships between classes). *Aspects* in AspectJ are then defined in a similar manner as Java classes, but they add joint-points, advices and inter-type declarations. They are the unit in which crosscutting concerns are modeled in AspectJ.

AspectJ places aspect specifications along side the class hierarchy, and therefore ties specification to classes. In contrast, CF relates aspects to objects by means of the characteristics of messages it receives. This facilitates class-wide aspect definition in AspectJ, but grants more expressiveness and reusability to CF. Also, AspectJ's semantics are mainly compile-time, whereas CF handles both runtime and compile-time semantics.

2.5.2 EAOP

Event-based AOP[SD02] (EAOP) is a general framework for AOP in which aspects are modeled as events. In it, sequences of related events are defined by patterns that are matched at runtime in order to execute an action. Currently, EAOP provides a prototype tool implemented in Java.

In EAOP, aspects are defined as point-cuts and actions to be carried out when these point-cuts (and other conditions) are met. Point-cuts correspond to execution points in the source code (statements such as method invocations); while actions state *what* must happen at those point-cuts. The composition of the various aspects that can occur during the execution of the program is clearly defined by the user through a configuration file.

This approach to AOP is similar to CF in its interpretation of aspects; for both processes have a matching and an execution part. This allows for a wider control on aspect execution than that provided by frameworks in which the matching process is fixed, like AspectJ 2.5.1.

2.5.3 Aspect C♯

Aspect C♯[Kim02] is a more or less straight forward port of the concepts of AspectJ to the C♯ language. It ports the same concepts, although it uses several facilities provided by the .Net platform, such as XML, for its processing.

The Aspect C♯ model makes a clear distinction between the "business" code and the "aspect" code by placing the latter in a separate file in the form of an XML document. It also supports the AspectJ model of joint-points, as well as static (inter-type definitions) and dynamic (advices) crosscutting.

Aspect C♯ compares with CF in the same way as AspectJ. One notable difference is that due to Compose*'s runtime implementation, it is applicable to all languages that run in the .Net platform; whereas the Aspect C♯ implementation relies on a preprocessor that acts on the source code, thereby binding it to the C♯ language.

Chapter 3

The Compose* System

3.1 Requirements

At the beginning of the Compose* Runtime project, certain features and characteristics were expected from it. Which filter types were to be implemented and how, and how should the runtime interact with the underlying platform. These features are translated into the functional and non functional requirements specified in the following two sections.

Its important to note what falls out of the scope of the Compose* Runtime. It will not implement the *weaving* part of CF and it will not implement a message interception mechanism. It must, however, be designed in a manner that facilitates these foreseeable evolutions.

3.1.1 Functional Requirements

- **The runtime must permit the addition and removal of filter modules at runtime.**

  This ties Compose* to a runtime design and rules out a compile time approach. This makes Compose* more similar to the previous implementation of Sina/ST than to ComposeJ (see 2.2.2).

- **The filters to be implemented are:**

  - **Dispatch,**
  - **Send,**
  - **Wait,**
  - **Error and**
  - **Meta**

  There are other kinds of filters, such as the Real-Time filter[ABvSB94], but these are the most common.

- **The runtime must provide support for *Filter Modules.***

There is no support for the concern construct and superimposition features such as the ones presented in ConcernJ [Car01]

- **.Net objects must be able to interact with Compose*.net objects by:**

  - **receiving messages from compose*.net objects**
  - **sending messages to compose*.net objects.**

  This assures that the Compose* system will integrate well with legacy code.

- **All .Net objects are suitable for imposition, provided that they conform with the given filter interface.**

3.1.2 Non Functional Requirements

- **The runtime must be developed within the .Net framework, using j♯ as a language.**

  These permits a possible port to the Java platform.

- **The runtime architecture will be implemented as an interpreter.**

  By implementing the runtime as an interpreted a dynamic design in which the functional requirements (in particular the addition and removal of filter modules in runtime) are easily fulfilled.

- **The Compose*.net runtime must maintain dependencies with the framework libraries at a minimum to facilitate the porting of the source code to the Java platform. In order to do this, the design must me both modular and explicit.**

  This also facilitates the possible port to the Java platform by placing a wrapper around the platform-specific library objects (such as data types and reflection facilities) so that can be easily replaced when changing platforms.

3.1.3 Challenges

Several challenges present to the development of the Compose* runtime. Some are related to the platform, others to the requirements stated before, while others from the CF model itself. All these must be addressed if the project is to be successful.

- **Behaviour Injection**

  As discussed in 2.2, the original name of composition filters was interface predicates. This is because the appliance of filters on top of objects changes their interface. For example, given a logging filter L, when applied to an object A, can change the interface by adding a isLogging() method. This operation will change the declared interface of object A, altering the type of the object.

  In compile time based implementations of the Composition Filters such as ComposeJ[Wic99] and ConcernJ [Car01] this poses no problem. When a filter is imposed over an object, this is compiled again, as are all other objects referencing to A, therefore adjusting to the interface changes.

  The runtime implementation of Composition Filters done in the past, Sina/st [Koo95]also avoided this problem because it was implemented on top of a *dynamically typed* language (Smalltalk). By employing the `messageNotUnderstood` mechanism, objects can handle messages that do not correspond to methods they implement. This allows the transparent mutation of the interface of the object.

  However, the type system that exists within the .Net platform is static. This is enforced at IL (bytecode) and cannot be bypassed without modifying the Common Language Runtime. This presents the problem of how to implement dynamic interfaces in a static typed language. Even if the Common Language Runtime is fooled into allowing dynamic types, the problem of how to use these dynamic types from statically typed languages such as C♯ remains, because the compiler must be aware of the dynamic nature of the Compose* objects, and forego checking on method calls to those objects.

- **Message Interception**

  Message interception is one of the main issues to deal with when implementing the CF model, because it is the most natural way of implementing the concept of filtering messages. This can be achieved in various ways, for example modifying the platform's CLR so that it catches method invocations; weaving either at source code level or IL level.

  However, taking into account the functional requirement that states that all Compose* objects can receive messages from any other object in the platform, not all alternatives are viable. Source code weavers are discarded because they are language dependant, and they are not useful on pre-compiled libraries whose source code is not known. The useful approaches left are IL weaving and CLR modification.

- **Platform Independence**

  One of the non functional requirements states that the dependency between the Compose* runtime and the underlying platform must be kept to a minimum. Nevertheless, for the Compose* runtime to work, facilities such as message reflection and dynamic invocation must be used. This, of course, hampers the desired feature of platform independence.

3.2 Design of the Compose* Runtime

The architecture of the Compose* Runtime is composed of three main parts: the message interception layer, the filter specification interpreter and the core runtime. All these parts communicate through well defined interfaces in the manner described in figure 3.2. In it, an object A sends a message m to a Compose* object B who has a filter module attached to it. The message is caught by the message interception layer, and delegated to the core runtime. The core runtime, depending on the nature of the filter module attached to B, uses the filter specification interpreter, and access a variety of objects which may or not include the original intended *B* object. Finally, the message is returned to the message interception layer, and back to *A*.

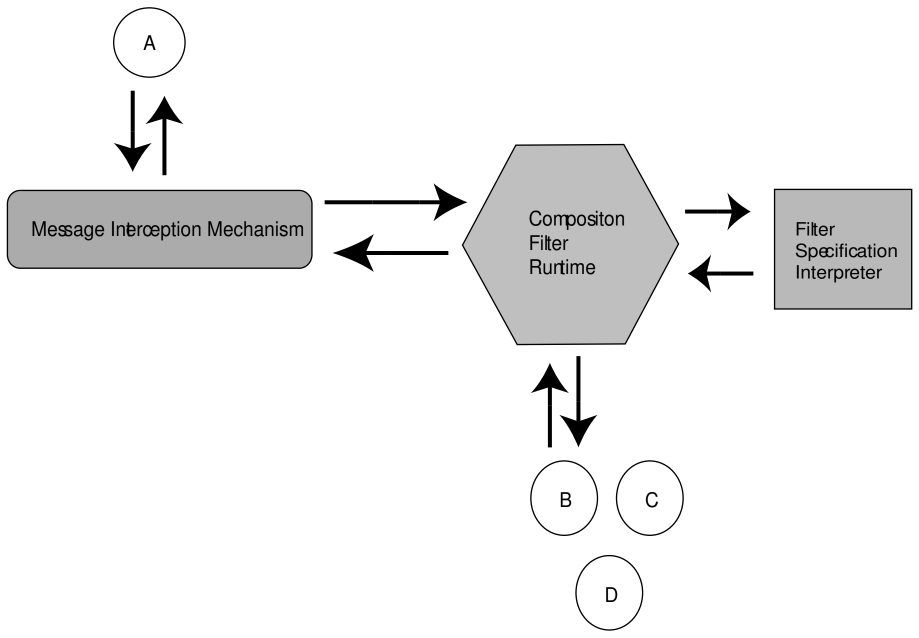
A
Message Interception Mechanism
Compositon Filter Runtime
Filter Specification Interpreter
B
C
D

Figure 3.1: Simplified architecture of Compose*Runtime

3.2.1 The Message Interception Layer

While the development of the message interception layer was not one of the original requirements of the project, it proved necessary for the correct implementation of the runtime. The message interception layer must provide means to catch method invocations at some point between the sender and the receiver object as transparently as possible while offering information about the invocation. At minimum, for the filtering process to be performed, this information must include:

- message identifier
- argument list
- target object

Other information such as return type expected and sender the instance, can be of use, but are not required.

Given these features, an interface was created to serve as entry point to the core runtime. Viewed in figure 3.2.1a, the MessageInterceptionListener interface is implemented with in the Runtime by a class that takes care of receiving intercepted messages and provides a return value. In this way, the Runtime is isolated from the particular message interception mechanism.

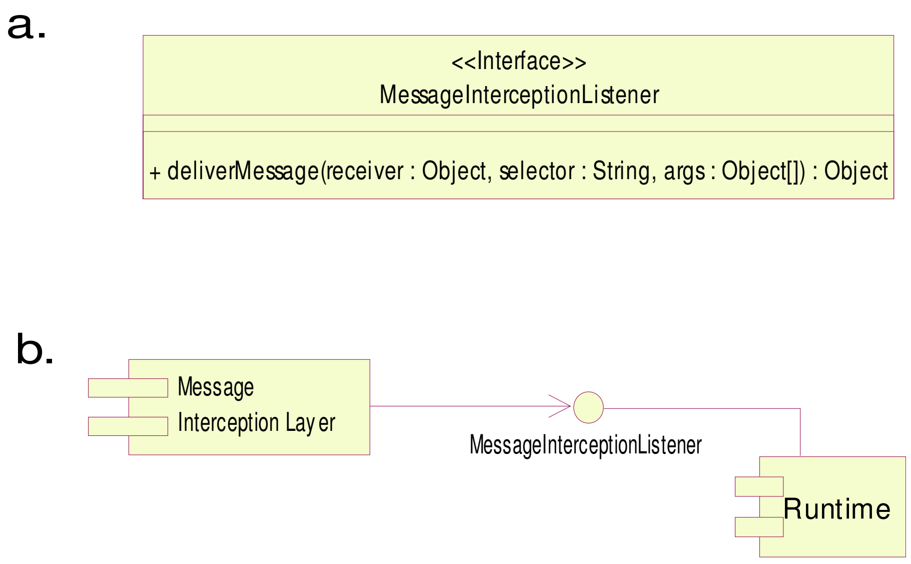
a.
<<Interface>>
MessageInterceptionListener
+ deliverMessage(receiver : Object, selector : String, args : Object[]) : Object
b.
Message
Interception Layer
MessageInterceptionListener
Runtime

Figure 3.2: a. Message interception interface b. Usage of interface

As mentioned in section 3.1.3, there are various alternative ways in which to implement the message interception component. However, given that message interception is not part of the scope of this project, an interception feature found in the .Net platform was chosen. This will be discussed in detail in section 3.3.1.

3.2.2 The Core Runtime

The general design of the core of the runtime is very similar to the CF model. In it a class, `ObjectManager`, controls messages incoming to the `ComposeStarObject` as defined by a `FilterPolicy`. This policy states how to pass the message through the various `FilterModules` and `Filters` within them. See figure 3.2.2.

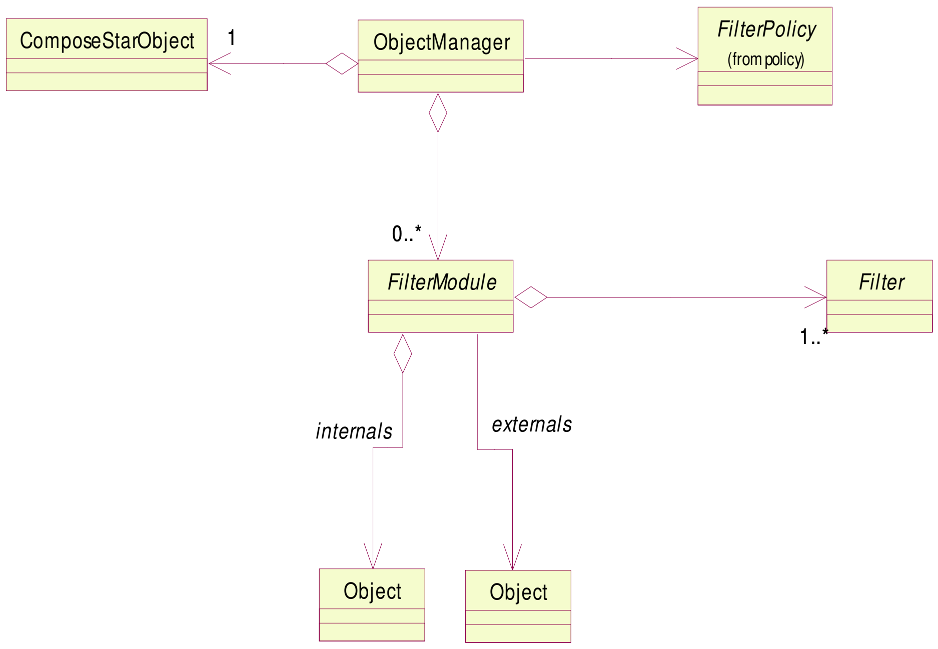

Figure 3.3: Simplified layout of the Runtime

The Object Manager

Each object whose messages can be filtered has an object manager assigned. This object manager is responsible for receiving and providing a return value for the messages that are sent to its inner object; also it is the point of contact between the Runtime and the Message Interception Layer.

The Object Manager holds the filter modules that are attached to its inner object. It passes each message through the filter modules one by one checking that each accepts the message. If one of the filter modules rejects a message, an exception is thrown. The Object Manager is not responsible, however, for *how* the message is passed through an individual filter module; that task is delegated to a Filter policy.

Filter and Filter Modules

Filter Modules implement aspects in the CF model. They contain filters and objects; as well as define a number of methods and conditions referenced by the filters they contain. The FilterModule class defines the common behaviour for the concrete FilterModules; such as the resolution of conditions and methods. The responsibility of resolving conditions is specified in the `ConditionResolver` interface.

Concrete subclasses of FilterModule, only need to specify the code relevant to the aspect they model. For example, in figure 3.4 a filter module that locks a certain object is modeled by the LockingFilterModule class. This class contains methods specific to the locking concern, `lock`,`unlock` and `isLocked`. `Lock` and `unlock` are mapped to regular methods of the FilterModule, whereas `isLocked` is mapped to a condition. This corresponds to the *implementation* part of the CF model.

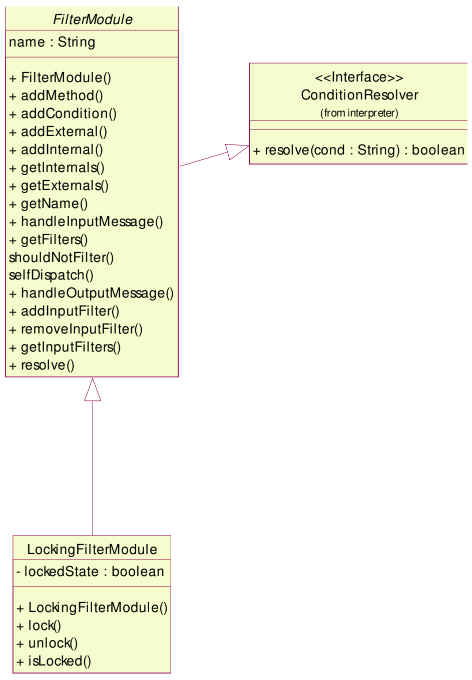

Figure 3.4: FilterModule class definition

As mentioned in section 2.2.1, the types of filters are differentiated by the action taken when a message is accepted or rejected. Therefore, new concrete filter classes need only to implement what action to take in case of rejection or acceptance of a message in the methods `acceptAction` and `rejectAction` see figure 3.5. These actions are modeled as instances of the Command pattern in the `ComposeStarAction` hierarchy so that they can be executed as defined by the Filter Policy.

If the filters actions are simple, like throwing an exception, it is sufficient to implement those methods; however, for more complex behaviour, such as the one given by the meta filter, changes to the way filter modules are necessary. For this reason a Filter Policy mechanism is provided.

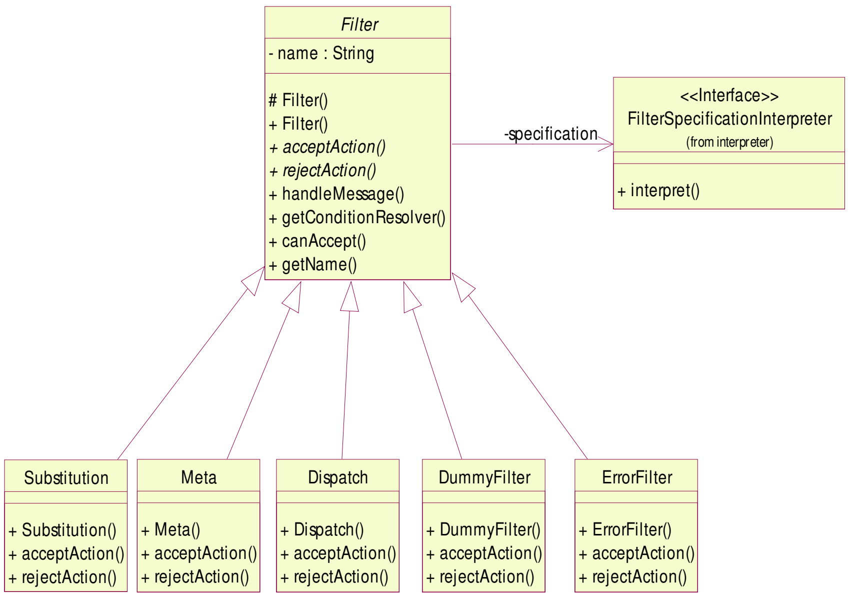

Figure 3.5: Filter class definition

Filter Policy

Filter policies dictate how are the messages passed through the filters within a filter module. They are modeled as a strategy pattern, which is invoked by the ObjectManager. The Policy at the end of the execution return a PolicyExecutionResult Object that contains information such as if the message was accepted by any of the filters in the filter module, or if the message should be returned to the sender because a return value has been produced.

The `FilterPolicy` is used, in conjunction with the `Filter` and `ComposeStarAction` classes to add new filter types to the system without breaking the existing code.

3.2.3 Specification Interpreter

The Specification of the Filter module, is an instance of the Interpreter pattern which both contain and interpret the specification for a given instance of a filter. In figure 3.6 The two main parts of the specification, condition and pattern, are connected by a `FilterInitializer`. Every class in the specification contains a method `interpret` that gives a value to the sub-expression that the class represents.

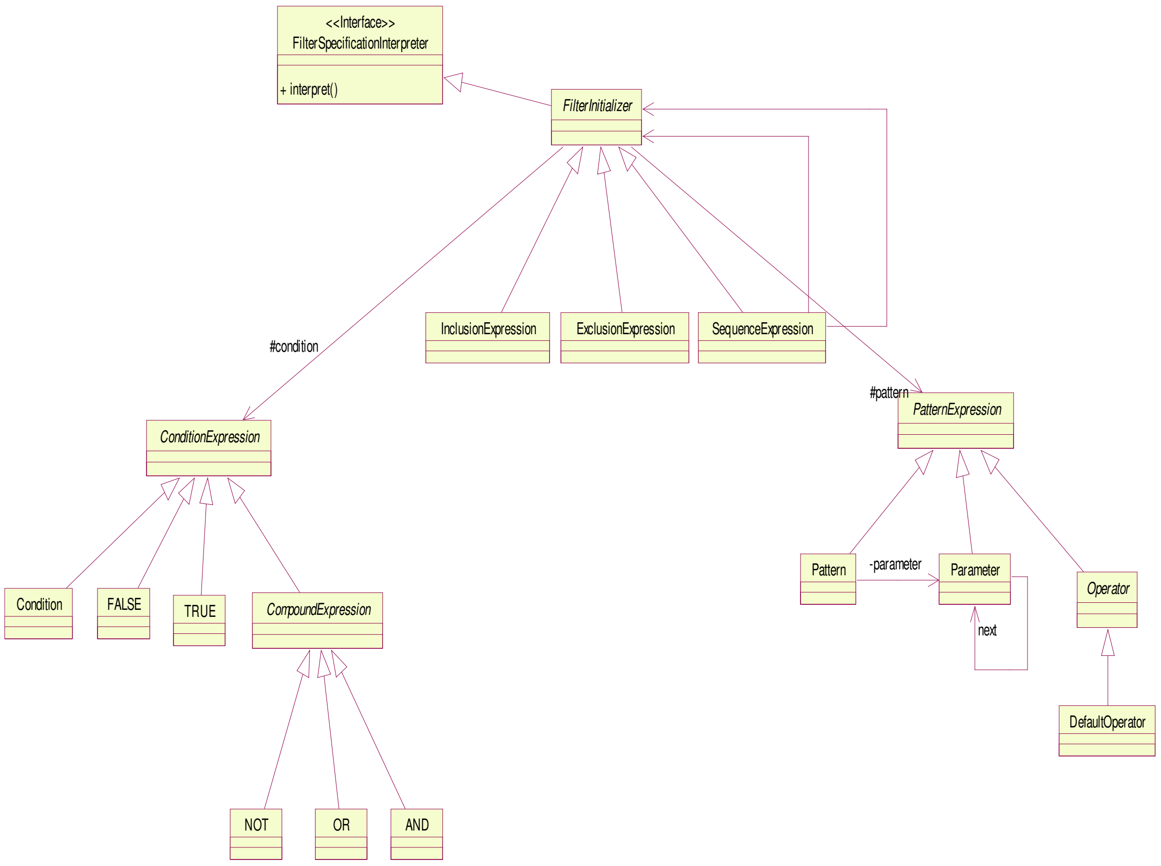

Figure 3.6: Specification Structure (some associations removed for convenience)

3.2.4 Challenges

In this section some of the challenges identified in section 3.1.3 are solved at a design level.

Integration of CF to .Net

To reduce the coupling between the .Net platform and the Compose* Runtime, wrapper classes to the platform specific libraries used are provided. These include wrappers for Data Types (queues, lists and dictionaries) as well as to the reflection and dynamic invocation classes.

The dynamic Interface problem - Behavior Injection

In order to resolve the dynamic interface issue produced because of the requirement to be able to change filter modules at runtime, several approaches were evaluated.

Looking closer at the problem, there are two main issues to deal with: how to compile classes that reference to filter module methods, and how to add and remove this methods from filtered objects. The second problem is easily solved with the help of the Message Interception System. Since the message is stopped on the way to the object, any other method can be invoked without the sender realizing the change. In this way, an object can appear to have many methods whose implementation can reside somewhere else [1].

However, defeating the compiler can be more difficult. One approach could be to modify the compiler into allowing for the calls to be made; nevertheless, since there are many languages that compile to the .Net platform, this approach is not viable. Another approach is to use the Metadata embedded into each .net library to *fool* the compiler into linking to non existing methods. This would be possible by statically analyzing the specification of the filters in the system, and altering the

[1]Given that the runtime implements this kind of behaviour only for FilterModules, there are restrictions that avoid problems with the `self` binding. In principle, the FilterModule methods do not reference inner object's instance variables; they can only reference the FilterModule internal and external objects.

library's metadata. This can be troublesome due to the fact that unexpected changes to the internal structure of the libraries (caused, for example, by a new .net platform version) would break the system.

An intermediate solution is provided. A class containing the union of the interfaces defined by the filter modules in the system is created with a default implementation for each method (the throwing of a runtime exception, a sort of message not understood). All objects that are to be filtered must extend this class. When a method belonging to a filter module placed on an object is invoked, the message interception layer catches the message, invokes the correct method in the filter module, and returns. If a method belonging to a filter module **not** placed on an object is invoked, a runtime exception is thrown.

3.3 Implementation of the Compose* Runtime

3.3.1 Message Interception Implementation

A provisional implementation of a Message Interception Layer for the Compose* Runtime is provided. It is based on the .Net component model, and uses undocumented features of the platform. For this reason, other alternatives should be researched.

The current implementation of the provides transparent, receiver-side message interception. This has as an advantage that no modifications to the sender object are necessary. The interception mechanism presents information about the target of the message, the name of the message and its arguments. It also allows for the interception of public instance variables (attributes) as well as constructor calls.

.Net Component Framework Contexts

The base of the message interception implementation lies in the COM framework provided by .Net. Different components in an application can request runtime services from the COM, for example synchronization, or transactions. Components with compatible service requirements are grouped by in *contexts*. As each class in the system can require different services, within a single process there can be several context separating incompatible components from each other.

Objects inside a same context can communicate freely with each other. But when cross-contexts reference are needed, lightweight proxies are constructed by the platform to handle the change from one contexts requirement to the next.

In the .Net platform, objects can be defined within property-based context; this means that objects with the same properties are placed on the same context. These properties may be COM attributes (such as synchronized). Objects that do not use COM attributes at all reside in the *default* context.

In .Net, the cross context communication is implemented through the principle of channels used for remote method invocation. In it, a message that is sent to a remote object is passed through a series of sinks in an implementation of a chain of responsibility. Each sink does a part of the process, for example marshaling of the arguments of the message.

Context-based interception

By using the undocumented classes `ContextAttribute` and `IContextProperty` it is possible to insert custom message sinks in the channel used to manage cross-context communication. To illustrate how this is done, suppose that it is needed to intercept messages that enter the class A. In order to do this, the following

1. a custom `InterceptAttribute` must be created by subclassing `ContextAttribute`.

2. a custom `InterceptProperty` must be created by implementing the `IContextProperty`. Also it must implement the `IContributeObjectSink`

3. the class A must subclass the `ContextBoundObject` class, and it must be annotated to have the Intercept attribute defined above in the class' metadata.

Once this is done, each time an instance of the class A is created, the platforms asks the InterceptAttribute if the current context is ok for the new instance. If its not, the platform asks the attribute to provide the properties of the instance. The

Attribute returns an instance of the Intercept Property that is also a Object Sink contributor; the platform then asks the property for a custom message sink, and inserts it in the cross-context channel for that object. From then on, all messages sent to the object from outside the context will pass by the message sink provided by the property. Figure 3.7 illustrates how this is used in the Compose* system by encapsulating each ComposeStarObject in a different context.

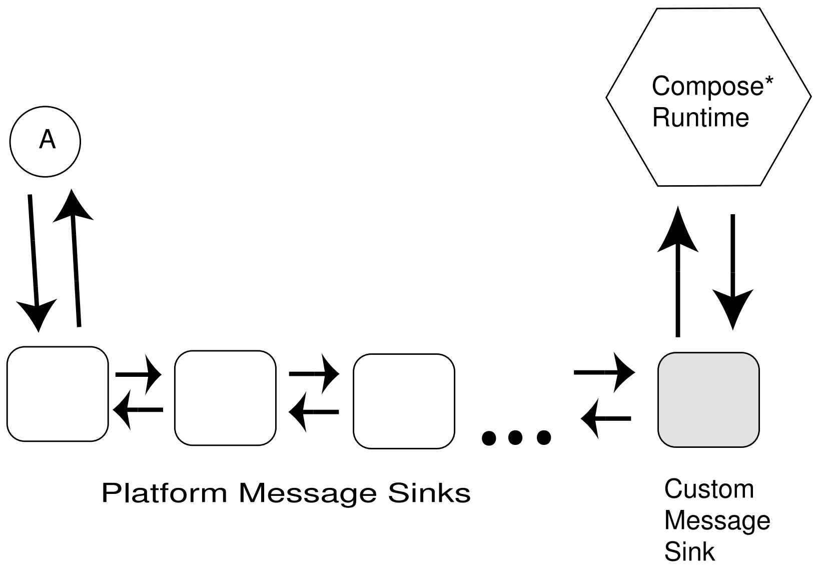

Figure 3.7: Message interception chain

3.3.2 Implementation of the Filters

Filters, as designed, are defined by the actions they take when accepting and rejecting a message. However, some of the actions affect the control flow of the filtering of the other filters; for example a meta filter can reify a message, and send it to complete the filtering process, only to reify it again when the return value is to be sent back to the original sender object. These filters also require changes in the policy by which they are executed. The particular implementation of the three filters is discussed below.

Dispatch Filter

The dispatch filter upon acceptance, redirects the message received to a different object. This is carried out when the `DispatchAction` that is generated by the filter is executed. When this action is executed, it terminates the filtering process.

Upon rejection, the dispatch filer produces a `ContinueToNextFilterAction`.

Error Filter

The error filter upon acceptance produces a `ContinueToNextFilterAction`. Upon rejection, the error filter produces a `ErrorAction` that when executed throws a Runtime exception that contains information about the expression that rejected the message.

Meta Filter

The meta filter is the most complex of the three because its actions affect how the rest of the filtering process is carried out. When the Meta Filter accepts a message, a `MetaAction` containing the message and the ACT object that will manipulate the reified message. When this action is executed, the message is reified, and sent to the ACT. When the ACT returns, the Reified message is queried to see whether the message was *fired*, *replied* or *sent*. Depending on the action taken on the ACT, the `MetaAction` will:

Fire: When a reified message is fired, it is sent to the next filter, and continues its normal filtering path.

Reply: When the message is replied, a return value is provided for it, and the message exits the filtering process and its return value sent back to the original sender.

Send: When the reified message is sent, the message continues its normal filtering path, only that when the filtering process is over, and a return value for the message has been established, the message must be reified again so that the ACT can manipulate the return value. This is implemented with the .Net version of function pointers, `delegates`. In this way, the ACT provides a callback method to be invoked with the reified message as a parameter when the message is on its way back to the sender. If there are multiple Meta filters in the filter module, their callbacks are collected in a stack, to preserve the correct invocation order.

Chapter 4

Conclusion

4.1 Evaluation

During the start of the development of the Compose* runtime a set of requirements were identified. Also, along the design and implementation process, several challenges were presented. In this section, an evaluation of how well were these goal reached is made.

4.1.1 Functional Requirements

- **Addition and removal of filter modules at runtime**

  This requirement was met. Filter Modules can be constructed and added to objects at runtime through the ObjectManager class. Also, existing modules can change their inner structure at runtime, due to the fact that they are interpreted on a *per message* bases.

- **Implementation of the Dispatch, Send, Wait, Error and Meta filters**

  This requirement was partially met. The Dispatch and error filters were implemented directly from the CF model without adaptation. The Meta Filter was modified so that the *send* operation on reified messages is implemented through re-entrant code; however, the semantic difference to the original CF model is minimal. The Send and Wait Filters were not implemented. The Send filter could not be implemented due to restrictions of the Message Interception Layer that prevent the interception of outgoing messages; therefore output filters, such as the Send filter, cannot be implemented. The Wait filter was not implemented due to time restrictions.

- **Implementation of the Filter Modules**

  The implementation of the Filter Modules is very close to the CF model. Internals, externals, conditions, methods and filters are fully implemented; however, pseudo-variables defined in the CF model were not implemented fully. Only the `inner` pseudo-variable was included.

- **Interaction between the .Net platform and Compose***

  Thanks to the message interception system, communication to and from Compose* objects is transparent.

- **Suitability of filter addition to all .Net Objects**

  This is partially met. Due to the solution given to the Behaviour Injection challenge, all objects that are to be filtered must extend a given object. This represents some inconvenience when programming new applications, but it can be a big problem when trying to apply filters to legacy code.

4.1.2 Challenges

At the end of the Requirements, in section 3.1.3, some challenges presented by the development of the tool were identified. In this section, the way in which the were overcome is evaluated.

- **Behaviour Injection**

  The way in which this challenge was overcome has both advantages and drawbacks. On one hand, the problem is resolved using object oriented constructs, and the solution allows for complete behaviour injection for a given set of filter modules. No modifications are needed on the compiler or CLR. On the other hand, it requires that all objects extend `ComposeStarObject` which can difficult the design of the application, and it can create problems when adapting existing code to Compose*. Also, since the interface of the `ComposeStarObject` is calculated at compile time, the sets of applicable module cannot be changed at runtime, that is filter modules can only be constructed at compile-time.

- **Message Interception**

  The message interception layer implemented has various advantages. It is transparent from the point of view of the client object, and it does not require complex source code manipulation on the receiver object (it is sufficient to extend the `ComposeStarObject`. The mechanism used provides the basic information needed for the filtering process that is message name, target object and message arguments. However, the performance of the application is affected by the use of this mechanism due to the context-crossing process; which is executed even when there are no filter modules placed on the object. Also, since the mechanism relies on unsupported features of the .Net platform, it could disappear or change in future versions of the .Net runtime.

- **Platform Independence**

  To achieve a degree of platform independence, all platform specific references were enclosed on wrapping classes, and grouped together in a specific namespace. Also, the J♯language was used to code most of the application so porting to the Java platform can be achieved. The only platform specific construct used in the Compose* are function pointers as a way to specify callback methods for the implementation of the Meta filter(see 3.3.2).

  Although no attempt was made to pass the source code through a Java compiler, Java-specific tools such as Javadoc were successfully used, hinting that, at least syntactically, the program complies with Java standard.

4.2 Conclusion

A Runtime based implementation of the Composition Filters Model was designed and implemented on the .Net platform. This implementation takes advantage of the language independence features present in the .Net platform to offer a single Aspect framework that provides dynamic crosscutting to all languages that compile to the .Net framework. By being implemented in the J♯ language and wrapping platform dependant references the system eases a future port to the Java platform.

The evaluation of the goals achieved and challenges overcome, shows that the Compose* runtime, while being a first implementation, is powerful enough to serve as a starting point for a larger project that cover the next generation of the Composition

Filters. Still, the evaluation also shows that there is much work to be done.

4.3 Future Work

This section describes the open issues and proposed improvements to the Compose* system.

- Implementation of the lacking filter Wait. In order to do this, the Object Manager must be made thread safe.
- Support for the remaining pseudo-variables defined in the CFmodel. Ways to implement this are discussed in [Wic99] from a compile-time approach.
- Due to issues of J♯'s implementation of the reflection libraries, there are problems when handling primitive types in arguments. This restricts the processing that can be applied inside a Meta filter.
- A more fitting way to solve the behaviour injection problem should be researched.
- A preprocessor that parses the CFdefinitions to Compose* runtime classes is being developed. This preprocessor must also generate the `ComposeStarObject` class containing the methods defined by all the filter modules to implement behaviour injection.
- Optimization based on custom FiliterPolicies can be researched. These policies can be used to reorder the filters using runtime information.
- A trial port of the Compose* runtime to the Java platform to take advantages of the new features of the upcoming Java JSE 1.5, such as generics, would also be interesting.

Bibliography

[ABvSB94] M. Aksit, J. Bosch, W. v.d. Sterren, and L. Bergmans, *Real-time specification inheritance anomalies and real-time filters*, Proc. ECOOP 1994, LNCS 821, 1994, pp. 386–407.

[BA] Lodewijk Bergmans and Mehmet Askit, *Composing multiple concerns using composition filters*.

[Car01] Patricio Salinas Caro, *Adding systemic crusscutting and super-imposition to composition filters*, Master's thesis, Ecole des Mines de Nantes - France and University of Twente - The Netherlands, 2001.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *An overview of AspectJ*, Proc. ECOOP 2001, LNCS 2072 (Berlin) (J. L. Knudsen, ed.), Springer-Verlag, June 2001, pp. 327–353.

[Kim02] Howard Kim, *Aspect c♯*, Master's thesis, Trinity College, Dublin, September 2002.

[KJS00] M. Kande, J.Kienzle, and A. Strohmeier, *From AOP to UML: Towards an aspect-oriented architectural modeling approach*, Tech. report, Swiss Federal Institute of Technology Lausanne, 2000.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-oriented programming*, Proceedings European Conference on Object-Oriented

Programming (Mehmet Akşit and Satoshi Matsuoka, eds.), vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[Koo95] Piet S. Koopmans, *On the definition and implementation of the sina/st language*, Master's thesis, Universiteit Twente, 1995.

[SD02] M. Südholt and R. Douence, *A model and a tool for event-based aspect-oriented programming*, Tech. report, Ecole des Mines de Nantes, 2002.

[TO01] Pery Tarr and Harold Ossher, *Using multidimensional separation of concerns to (re)shape evolving software*, Communications of the ACM (2001), 43–50.

[Wic99] J.C Wichman, *Composej: the development of a preprocessor to facilitate composition filters in the java language*, Master's thesis, University of Twente, December 1999.

Appendix A

A Compose* Example

A.1 The Locking Filter

This example shows how to use the Compose* runtime to implement a filter that will block messages that enter an object. One possible application of such a filter can be seen in drawing applications in which is possible to "lock" figures to prevent alteration.

The source code for `LockingFilterModule` is presented in figure A.1. The `LockingFilterModule` class contains a method to lock the object, one to unlock it and one to test the lock. In the constructor of the class, resides the code that defines the structure of the FilterModule. It is composed of two methods: `lock` and `unlock`; one condition `isLocked` and two input filters. The first filter is an Error Filter that uses the `isLocked` condition so that if the object is locked, all messages are rejected; while if the object is not locked, all messages are accepted. The second filter is a Dispatch filter that receives all messages accepted by the ErrorFilter, and executes them.

To place an instance of `LockingFilterModule` over a given object, the object's manager must be obtained, see figure A.2. Once the filter module is added to the ObjectManager, all messages will pass through the filter module.

```
public class LockingFilterModule extends FilterModule {
    private boolean lockedState;

    public LockingFilterModule(String name) {
        super(name);
        lockedState = false;
        addMethod("lock");
        addMethod("unlock");
        addCondition("isLocked");

        // precondition : Error = {! isLocked => *}
        Filter f =
            new ErrorFilter("precondition",
            new InclusionExpression(new NOT(
                                    new Condition("isLocked")),
                                    new Pattern("*")), this);


        //invoke : Dispatch = { true => inner.*}
        this.addInputFilter(new Dispatch("invoke",
                                 new InclusionExpression(new TRUE(),
                                 new Pattern("inner", "*")),this));
        this.addInputFilter(f);
    }


    public void lock() {
        lockedState = true;
    }


    public void unlock() {
        lockedState = false;
    }

    public boolean isLocked() {
        return lockedState;
    }
}
```

Figure A.1: LockingFilterModule implementation in J♯

```
try {
  LockingFilterModule lock = new LockingFilterModule();
  ObjectManager.getObjectManagerFor(superImposee).
                                         addFilterModule(lock);

  superImposee.lock();
  superImposee.foo();

  } catch (ErrorFilterException efe) {

     superImposee.unlock();
     superImposee.foo();

  }
```

Figure A.2: Imposition of the LockingFilter module on an object

Appendix B

Testing Guidelines

B.1 Introduction

For the development of the Compose* system[1], starting with the runtime module, a Unit Testing strategy must be followed. For this purpose several external libraries are available, some of them include:

- dotUnit [http://dotunit.sourceforge.net/]
- Nunit [http://nunit.org/default.htm]
- csUnit [http://sourceforge.net/projects/csunit/]
- .NetUnit[http://sourceforge.net/projects/dotnetunit/]
- ...

All of which follow the JUnit [http://www.junit.org/] framework. Since most of the frameworks do not stray far from the original framework, they are in essence similar, however subtle differences in the specific APIs prevent easy migration between them.

For the development of the runtime module of the Compose* system, the dotUnit was chosen. There is no particular reason for choosing this framework over the rest

[1]This document was created for the sourceforge web page to try and establish project-wide unit testing guidelines

other than it is close to the JUnit implementation, and that it is easy to install -not requiring administrator rights under Win2000-.

For a general introduction to dotUnit, see [http://dotunit.sourceforge.net/tutorial.htm]

B.2 Usage in Compose*

B.2.1 Naming standards

- Tests for a class must be placed in a single file.

  The name of the file is "Test" plus the name of the class that it tests.

- Test files must be grouped in Suites by namespace.

  A group of tests fixtures for a given namespace must be placed with in that test as a sub namespace called tests. For example, all tests for the namespace called

  `dotNetComposeStar.runtime.interpreter`

  must be placed on a namespace

  `dotNetComposeStar.runtime.interpreter.tests`

  To bind a tests in a Suite see this j snippet:

```
TestSuite s = new TestSuite("runtime and sub-package");
s.AddTestSuite(new TestFilter("foo").GetType());
s.AddTestSuite(new TestFilterModule("foo").GetType());
```

  which binds the Tests for the Filter and FilterModule classes.

- Test suites for each namespace must include the tests for its sub namespaces, such that the test suite for dotNetComposeStar.runtime must include the one for dotNetComposeStar.runtime.interpreter. This is accomplished by:

  Note that when adding a suite the AddTest method is used, and when adding a test, the AddTestSuite method is used.

```
   s.AddTest(new InterpreterSuite().get_Suite());
```

B.3 Configuring and running a project for dotUnit in Visual Studio

Once dotUnit is installed, to use it on a project you must include it as a reference. This is done by

1. With the project open, under the "Project" menu, select "add Reference..."
2. Select a "Browse", and look for "dotUnit.dll" under the directory in which dotUnit was installed
3. Finally, select the library, and add it to the references.

To run a test suite:

```
   dotUnit.GUI.GUIRunner.Run(new dotNetComposeStar.
                     runtime.tests.RuntimeTestSuite().get_Suite());
```

Appendix C

Source Code Documentation

C.1 Package dotNetComposeStar.runtime.interpreter

Package Contents *Page*

C.2 Interfaces

C.2.1 Interface ConditionResolver

Objects with the responsibility of resolving conditions.

Currently this responsibility lies in the FilterModule

Declaration

```
public interface ConditionResolver
```

Methods

- *resolve*

  `public boolean resolve( java.lang.String  cond )`

  - **Usage**
    - This method should decide if a condition identified by the string cond evals to true or false.
  - **Parameters**
    - `cond` - the string identifing the condition
  - **Returns** - the evaluation of the condition

C.2.2 Interface FilterSpecificationInterpreter

Intepreter of Filter Specifications.

Currently implemented in the FilterInitialization hierarchy as a Interpreter Pattern instance.

Declaration

```
public interface FilterSpecificationInterpreter
```

Methods

- *interpret*

  `public boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    - Interprets a FilterSpecification. It descides if a given message is accepted or not
  - **Parameters**
    - `m` - message that is being filtered
    - `context` - context of the interpretation
  - **Returns** - boolean true if the message is accepted, false otherwise

C.3 Classes

C.3.1 Class AND

AND operator for the condition part of a filter specification.

Declaration

public class AND
extends dotNetComposeStar.runtime.interpreter.CompoundExpression

Constructors

- *AND*

  `public AND(`
  `dotNetComposeStar.runtime.interpreter.ConditionExpression  exp1,`
  `dotNetComposeStar.runtime.interpreter.ConditionExpression  exp2 )`

- **Usage**
  - exp1 && exp2
- **Parameters**
  - `exp1` - ConditionExpression 1
  - `exp2` - ConditionExpression 2

Methods

- *interpret*

  `public boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    - interpretation(expression1) && interpretation(expression2)
  - **Parameters**
    - `m` - message being filtered
    - `context` - context of the intepretation
  - **Returns** - false if any of the interpretations is false, true otherwise

Methods inherited from class `dotNetComposeStar.runtime.interpreter.CompoundExpression`

(in C.3.2, page 55)

Methods inherited from class `dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*
  `public abstract boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    - Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.

- **Parameters**
  - `m` - message that is being filtered
  - `context` - context of the interpretation
- **Returns** - boolean the evaluation of the expression.

C.3.2 Class CompoundExpression

Super class of compount condition expressions (AND,OR, NOT)

Declaration

public abstract class CompoundExpression
extends dotNetComposeStar.runtime.interpreter.ConditionExpression

Constructors

- *CompoundExpression*

  `public CompoundExpression( )`

  - **Usage**
    - Default constructor

      since this class is used only to group compound expressions, it is empty.

Methods inherited from class `dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    - Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.

- **Parameters**
  - `m` - message that is being filtered
  - `context` - context of the interpretation
- **Returns** - boolean the evaluation of the expression.

C.3.3 Class Condition

Terminal expression that contains a String representing a condition in the current FilterModule specification.

Declaration

public class Condition
extends dotNetComposeStar.runtime.interpreter.ConditionExpression

Constructors

- *Condition*

  `public Condition( java.lang.String  selector )`

  - **Usage**
    - Constructs a condition based on its name.

      This name must be meaningful for the condition resolver that is selected at runtime. If not, an Invalid Condition Exception will be thrown.
  - **Parameters**
    - `selector` - the name of the condition

Methods

- *interpret*

  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

- **Usage**
  - Intepretation of the Condition

    This results on the invocation of a method that will return a boolean. this boolean will be the return value of this method. The invocation, however, is delegated to a ConditionResolver that is expected to be in the context of the interpretation under the key "ConditionResolver" (case sensitive)
- **Parameters**
  - `m` - message that is being filtered
  - `context` - that must contain the key specified before.
- **Returns** - the result of the condition evaluation.

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
  - **Parameters**
    - `m` - message that is being filtered
    - `context` - context of the interpretation
  - **Returns** - boolean the evaluation of the expression.

C.3.4 Class ConditionExpression

Parent class of the condition part of Filter Specifications It represent contidions and the logical expressions AND,OR,NOT,TRUE and FALSE

Declaration

public abstract class ConditionExpression
extends java.lang.Object

Constructors

- *ConditionExpression*
  `public ConditionExpression( )`
  - **Usage**
    * Constructs a Condition expression

Methods

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
  - **Parameters**
    * `m` - message that is being filtered
    * `context` - context of the interpretation
  - **Returns** - boolean the evaluation of the expression.

C.3.5 Class DefaultOperator

Sequencing operator for the Pattern part of a FilterSpecification.

When interpreted, the default operator evaluates the first pattern, if there is no match, it evals the second pattern.

Declaration

public class DefaultOperator
extends dotNetComposeStar.runtime.interpreter.Operator

Constructors

- *DefaultOperator*
  ```
  public DefaultOperator(
  dotNetComposeStar.runtime.interpreter.PatternExpression  exp1,
  dotNetComposeStar.runtime.interpreter.PatternExpression  exp2 )
  ```
  - **Usage**
    * Constructor of the default sequence operator.
  - **Parameters**
    * `exp1` - first expression
    * `exp2` - second expression.

Methods

- *interpret*
  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * intepretation(exp1) —— interpretation(exp2)
  - **Parameters**
    * `m` - message that is being filtered

    - `context` - context of the interpretation.
  - **Returns** - the evaluation of the two expressions.

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.Operator`

(in C.3.11, page 68)

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.PatternExpression`

(in C.3.15, page 74)

- *interpret*
  `public boolean interpret( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    - empty
  - **Parameters**
    - `m` -
    - `context` -
  - **Returns** - boolean

C.3.6 Class ExclusionExpression

Expression that represents the >operation

Declaration

---

public class ExclusionExpression
extends dotNetComposeStar.runtime.interpreter.FilterInitializer

Constructors

- *ExclusionExpression*
  `public ExclusionExpression(`
  `dotNetComposeStar.runtime.interpreter.ConditionExpression cond,`

```
dotNetComposeStar.runtime.interpreter.PatternExpression  pattern
)
```

– **Usage**
  * Constructs a new Exclusion Expression
– **Parameters**
  * `cond` - The condition part of the expression
  * `pattern` - The message pattern part of the expression.

Methods

- *interpret*

  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  – **Usage**
    * Interprets the exclusion

      That is, for the expression to accept the message, the condition expression must evaluate to TRUE, and the message must not match the pattern expression.
  – **Parameters**
    * `m` - message being filtered
    * `context` - the context of the intepretation.
  – **Returns** - if the message is accepted by the expression

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.FilterInitializer`

(in C.3.8, page 63)

- *interpret*

  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  – **Usage**

    - Interprets the message in the given context
  - **Parameters**
    - `m` - Message to interpret
    - `context` - context
  - **Returns** - boolean

C.3.7 Class FALSE

Condition expression FALSE

Declaration

public class FALSE
extends dotNetComposeStar.runtime.interpreter.ConditionExpression

Constructors

- *FALSE*

  `public FALSE( )`

  - **Usage**
    - Constructs a new false condition

Methods

- *interpret*

  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Allways evals to FALSE
  - **Parameters**
    - `m` - the message being filtered.
    - `context` - the context of the interpretation
  - **Returns** - FALSE

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
  - **Parameters**
    * `m` - message that is being filtered
    * `context` - context of the interpretation
  - **Returns** - boolean the evaluation of the expression.

C.3.8 Class FilterInitializer

FilterInitializer is the definition of a FilterSpecification.

it represents the expression between {} in a Filter

Declaration

public abstract class FilterInitializer
extends java.lang.Object
implements FilterSpecificationInterpreter

Constructors

- *FilterInitializer*

  `protected FilterInitializer( )`

  - **Usage**
    * Default constructor.

- *FilterInitializer*

  `public FilterInitializer(`

  `dotNetComposeStar.runtime.interpreter.ConditionExpression  cond,`

  `dotNetComposeStar.runtime.interpreter.PatternExpression  pattern`

  `)`

  - **Usage**
    - Constructs a filter initializer with its condition and pattern.
  - **Parameters**
    - `cond` - condition part
    - `pattern` - pattern part

Methods

- *interpret*

  `public abstract boolean interpret(`

  `dotNetComposeStar.runtime.message.Message  m,`

  `dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    - Interprets the message in the given context
  - **Parameters**
    - `m` - Message to interpret
    - `context` - context
  - **Returns** - boolean

C.3.9 Class InclusionExpression

Expression that represents the =>operator.

Declaration

public class InclusionExpression

extends dotNetComposeStar.runtime.interpreter.FilterInitializer

Constructors

- *InclusionExpression*

  ```
  public InclusionExpression(
  dotNetComposeStar.runtime.interpreter.ConditionExpression  cond,
  dotNetComposeStar.runtime.interpreter.PatternExpression  pattern
  )
  ```

  - **Usage**
    - Constructs a new Inclusion Expression out of the condition and pattern.
  - **Parameters**
    - `cond` - Condition part
    - `pattern` - Pattern part

Methods

- *interpret*

  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Interprets the InclusionExpression.

      That is, it interprets the logical condition part, and it matches the message against the pattern part. If both evaluate to true, the Inclusion Expression will evaluate to true also, otherwise, it will return false.
  - **Parameters**
    - `m` - Message that is being filtered
    - `context` - of the interpretation
  - **Returns** - if the InclusionExpression accepts the message.

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.FilterInitializer`

(in C.3.8, page 63)

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    - Interprets the message in the given context
  - **Parameters**
    - `m` - Message to interpret
    - `context` - context
  - **Returns** - boolean

C.3.10 Class NOT

NOT in a logical condition expression

Declaration

public class NOT
extends dotNetComposeStar.runtime.interpreter.CompoundExpression

Constructors

- *NOT*
  ```
  public NOT(
  dotNetComposeStar.runtime.interpreter.ConditionExpression  exp )
  ```
  - **Usage**
    - Constructs a NOT
  - **Parameters**
    - `exp` - expression to negate

Methods

- *interpret*

  `public boolean interpret( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - Interprets the expression by negating the value of the sub expression.
  - **Parameters**
    - `m` - Message that is being filtered
    - `context` - of the interpretation
  - **Returns** - a negation of the evaluation of the subexpression.

Methods inherited from class `dotNetComposeStar.runtime.interpreter.CompoundExpression`

(in C.3.2, page 55)

Methods inherited from class `dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*

  `public abstract boolean interpret( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
  - **Parameters**
    - `m` - message that is being filtered
    - `context` - context of the interpretation
  - **Returns** - boolean the evaluation of the expression.

C.3.11 Class Operator

Operator for patterns.

Introduced to keep the door open for operator other than the default operator " , ".

Declaration

public abstract class Operator
extends dotNetComposeStar.runtime.interpreter.PatternExpression

Constructors

- *Operator*
  ```
  public Operator(
  dotNetComposeStar.runtime.interpreter.PatternExpression  exp1,
  dotNetComposeStar.runtime.interpreter.PatternExpression  exp2 )
  ```
  - **Usage**
    * Constructs the binary operator.
  - **Parameters**
    * `exp1` - expression 1
    * `exp2` - expression 2

Methods inherited from class `dotNetComposeStar.runtime.interpreter.PatternExpression`

(in C.3.15, page 74)

- *interpret*
  ```
  public boolean interpret( dotNetComposeStar.runtime.message.Message
  m, dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * empty
  - **Parameters**
    * `m` -
    * `context` -
  - **Returns** - boolean

C.3.12 Class OR

Or expression on condition expressions

Declaration

public class OR
extends dotNetComposeStar.runtime.interpreter.CompoundExpression

Constructors

- *OR*
  ```
  public OR(
  dotNetComposeStar.runtime.interpreter.ConditionExpression  exp1,
  dotNetComposeStar.runtime.interpreter.ConditionExpression  exp2 )
  ```
  - **Usage**
    * Constructs an OR from the two subexpressions
  - **Parameters**
    * `exp1` - expression 1
    * `exp2` - expression 2

Methods

- *interpret*
  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * expression 1 —— expression 2
  - **Parameters**
    * `m` - message that is being filtered

        * `context` - context of the interpretation

    – **Returns** - the or of the interpretation of the two expressions.

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.CompoundExpression`

(in C.3.2, page 55)

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*
  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
    – **Usage**
        * Interpretation of the expression, accord the GoF pattern Interpreter.

          The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
    – **Parameters**
        * `m` - message that is being filtered
        * `context` - context of the interpretation
    – **Returns** - boolean the evaluation of the expression.

C.3.13 Class Parameter

Parameter on a Pattern expression.

Declaration

---

public class Parameter
extends dotNetComposeStar.runtime.interpreter.PatternExpression

Constructors

- *Parameter*

  `public Parameter( java.lang.String  messageElement,`
  `java.lang.String  identifier )`

  - **Usage**
    - Constructs a parameter
  - **Parameters**
    - `messageElement` - Key of the parameter
    - `identifier` - value of the parameter

- *Parameter*

  `public Parameter( java.lang.String  messageElement,`
  `java.lang.String  identifier,`
  `dotNetComposeStar.runtime.interpreter.Parameter  nextParameter )`

  - **Usage**
    - Constructs a list of parameters.
  - **Parameters**
    - `messageElement` - Key of the parameter
    - `identifier` - value of the parameter
    - `nextparameter` - next parameter in the list

Methods

- *getIdentifier*

  `public String getIdentifier( )`

  - **Usage**
    - Returns the value of the parameter
  - **Returns** - the identifier

- *getMessageElement*

  `public String getMessageElement( )`

  - **Usage**
    - Returns the key of the parameter

  - **Returns** - the messageElement

- *interpret*

  `public boolean interpret(`

  `dotNetComposeStar.runtime.message.Message  m,`

  `dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    * Interprets the Parameter element

      It adds the (messageElement, identifier) pair to the context and, interprets the next node in the list.
  - **Parameters**
    * `m` - Message that is being filtered
    * `context` - of the interpretation
  - **Returns** - true

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.PatternExpression`

(in C.3.15, page 74)

- *interpret*
  `public boolean interpret( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * empty
  - **Parameters**
    * `m` -
    * `context` -
  - **Returns** - boolean

C.3.14 Class Pattern

Terminal expression that represents patterns in a Filter Specification.

It desides if a message matches with the pattern, and it tries to bind the target and selector of the message to actual objects (pseudo variables, internals or externals)

Declaration

public class Pattern
extends dotNetComposeStar.runtime.interpreter.PatternExpression

Constructors

- *Pattern*

  `public Pattern( java.lang.String  selector )`

  - **Parameters**
    - `selector` -

- *Pattern*

  `public Pattern( java.lang.String  selector, dotNetComposeStar.runtime.interpreter.Parameter  p )`

- *Pattern*

  `public Pattern( java.lang.String  target, java.lang.String selector )`

  - **Parameters**
    - `target` -
    - `selector` -

- *Pattern*

  `public Pattern( java.lang.String  selector, java.lang.String target, dotNetComposeStar.runtime.interpreter.Parameter  p )`

Methods

- *interpret*

  `public boolean interpret( dotNetComposeStar.runtime.message.Message  m, dotNetComposeStar.util.Dictionary  context )`

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.PatternExpression`

(in C.3.15, page 74)

- *interpret*
  `public boolean interpret( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * empty
  - **Parameters**
    * `m` -
    * `context` -
  - **Returns** - boolean

C.3.15 Class PatternExpression

Expression that represent the pattern sub expression of a Filter specification.

Declaration

public abstract class PatternExpression
extends java.lang.Object

Constructors

- *PatternExpression*
  `public PatternExpression( )`
  - **Usage**
    * Default Constructor

Methods

- *interpret*
  `public boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

- **Usage**
  - empty
- **Parameters**
  - `m` -
  - `context` -
- **Returns** - boolean

C.3.16 Class SequenceExpression

grouping of two Filter Initialization Expressions so that one is evaluated before that the other one.

Declaration

public class SequenceExpression
extends dotNetComposeStar.runtime.interpreter.FilterInitializer

Constructors

- *SequenceExpression*

```
public SequenceExpression(
dotNetComposeStar.runtime.interpreter.FilterInitializer  exp1,
dotNetComposeStar.runtime.interpreter.FilterInitializer  exp2 )
```

  - **Usage**
    - Constructs a new sequence expression.
  - **Parameters**
    - `exp1` - expression 1
    - `exp2` - expression 2

Methods

- *interpret*

  `public boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  – **Usage**
    - Interprets the sequence

      It first evals the exp1, and if it does not accept the message, it evals the second one.

  – **Parameters**
    - `m` - Message
    - `context` - of the interpretation

Methods inherited from class
`dotNetComposeStar.runtime.interpreter.FilterInitializer`

(in C.3.8, page 63)

- *interpret*
  `public abstract boolean interpret(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  – **Usage**
    - Interprets the message in the given context

  – **Parameters**
    - `m` - Message to interpret
    - `context` - context

  – **Returns** - boolean

C.3.17 Class TRUE

True literal for the logic condition expressions.

Declaration

public class TRUE
extends dotNetComposeStar.runtime.interpreter.ConditionExpression

Constructors

- *TRUE*

  `public TRUE( )`

  - **Usage**
    - Constructs a true literal

Methods

- *interpret*

  ```
  public boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Interprets True
  - **Parameters**
    - `m` - message that is being evaluated.
    - `context` - of the evaluation
  - **Returns** - true

Methods inherited from class `dotNetComposeStar.runtime.interpreter.ConditionExpression`

(in C.3.4, page 57)

- *interpret*

  ```
  public abstract boolean interpret(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Interpretation of the expression, accord the GoF pattern Interpreter.

      The Condition expression is a logical, therefore its interpretation must return a boolean that contains the evaluation of the logical expression.
  - **Parameters**
    - `m` - message that is being filtered
    - `context` - context of the interpretation
  - **Returns** - boolean the evaluation of the expression.

C.4 Package dotNetComposeStar.exception

Package Contents *Page*

C.5 Classes

C.5.1 Class ComposeStarException

General Exception for the composeStar runtime.

Declaration

public class ComposeStarException
extends java.lang.RuntimeException

Constructors

- *ComposeStarException*
  `public ComposeStarException( )`
  - **Usage**
    * Default constructor

- *ComposeStarException*
  `public ComposeStarException( java.lang.String  message )`
  - **Usage**
    * Constructs a new ComposeStarException carrying a String as a message
  - **Parameters**
    * `message` - General message to send

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.2 Class ErrorFilterException

This exception models the error that comes from the rejection of a message by the Error Filter.

It is meant to be thrown during the message filtering process

Declaration

public class ErrorFilterException
extends dotNetComposeStar.exception.FilterException

Constructors

- *ErrorFilterException*

  `public ErrorFilterException( )`

  - **Usage**
    - Default constructor

- *ErrorFilterException*

  `public ErrorFilterException( java.lang.String  m )`

Methods inherited from class `dotNetComposeStar.exception.FilterException`

(in C.5.3, page 82)

- *getFilter*
  `public Filter getFilter( )`
  - **Usage**
    - Returns the filter in which the exception occurred.
  - **Returns** - Filter

- *setFilter*
  `public void setFilter( dotNetComposeStar.runtime.Filter  filter )`
  - **Usage**
    - Sets the filter in which this exception occurred for later retrieval.
  - **Parameters**
    - `filter` - The filter in which the exception occurred.

Methods inherited from class `dotNetComposeStar.exception.FilterModuleException`

(in C.5.4, page 85)

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.3 Class FilterException

General filter exception.

Contains information about the filter that generated the exception.

Declaration

public class FilterException
extends dotNetComposeStar.exception.FilterModuleException

Constructors

- *FilterException*

  `public FilterException( )`

  - **Usage**
    - Constructs a Filter Exception.

- *FilterException*

  `public FilterException( java.lang.String  caption )`

  - **Usage**
    - Constructs a Filter Exception with an accompanying message
  - **Parameters**
    - `caption` - the message that goes with the exception

Methods

- *getFilter*

  `public Filter getFilter( )`

  - **Usage**
    - Returns the filter in which the exception occurred.
  - **Returns** - Filter

- *setFilter*

  `public void setFilter( dotNetComposeStar.runtime.Filter  filter )`

  - **Usage**

    * Sets the filter in which this exception occurred for later retrieval.

  – **Parameters**

    * `filter` - The filter in which the exception occurred.

Methods inherited from class
`dotNetComposeStar.exception.FilterModuleException`

(in C.5.4, page 85)

Methods inherited from class
`dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`

- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.4 Class FilterModuleException

Exception at the filter module level.

This exception can occur, for example, when a call to a non existing condition is made.

Declaration

public class FilterModuleException
extends dotNetComposeStar.exception.ComposeStarException

Constructors

- *FilterModuleException*

  `public FilterModuleException( )`

  – **Usage**

    ∗ Default constructor

- *FilterModuleException*

  `public FilterModuleException( java.lang.String  message )`

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.5 Class FilterSpecificationException

Represents an exception occurred when interpreting the specification of a filter.

for example a message pattern pointing to a non existing internal or external.

Declaration

public class FilterSpecificationException
extends dotNetComposeStar.exception.ComposeStarException

Constructors

- *FilterSpecificationException*

  `public FilterSpecificationException( )`

- *FilterSpecificationException*

  `public FilterSpecificationException( java.lang.String  message )`

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`

- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.6 Class InvalidConditionException

Exception that deals with calls to invalid conditions (that is, conditions that are not defined in the enclosing FilterModule).

Contains information about the condition resolver used.

Declaration

public class InvalidConditionException
extends dotNetComposeStar.exception.FilterSpecificationException

Constructors

- *InvalidConditionException*
  `public InvalidConditionException( )`
  - **Usage**
    * Constructs a new Invalid Condition Exception
- *InvalidConditionException*
  `public InvalidConditionException( java.lang.String  message )`

- **Usage**
  - Constructs a new Invalid Condition Exception along with the accompanying message
- **Parameters**
  - `message` - the caption of the exception

Methods

- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    - Retrieves the Condition resolver that generated the error
  - **Returns** - the condition resolver

- *setConditionResolver*
  `public void setConditionResolver(`
  `dotNetComposeStar.runtime.interpreter.ConditionResolver`
  `cndResolver )`
  - **Usage**
    - Places the condition resolver that generated the error
  - **Parameters**
    - `cndResolver` - a class that resolves the Conditions

Methods inherited from class
`dotNetComposeStar.exception.FilterSpecificationException`

(in C.5.5, page 86)

Methods inherited from class
`dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.7 Class InvalidPatternExpressionException

Exception occurred while interpreting a pattern expression.

Declaration

---

public class InvalidPatternExpressionException
extends dotNetComposeStar.exception.FilterSpecificationException

Constructors

- *InvalidPatternExpressionException*
  `public InvalidPatternExpressionException( )`
  - **Usage**
    * Default constructor

- *InvalidPatternExpressionException*
  `public InvalidPatternExpressionException( java.lang.String message )`

Methods inherited from class `dotNetComposeStar.exception.FilterSpecificationException`

(in C.5.5, page 86)

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.8 Class MessageNotFilteredException

Exception thrown when a message traverses all the filters within a filter module and is not filtered by any of them.

It contains information about the message that was being filtered.

Declaration

public class MessageNotFilteredException
extends dotNetComposeStar.exception.FilterModuleException

Constructors

- *MessageNotFilteredException*

  `public MessageNotFilteredException( )`

- *MessageNotFilteredException*

  `public MessageNotFilteredException( java.lang.String caption )`

Methods

- *getComposeStarMessage*

  `public Message getComposeStarMessage( )`

  - **Usage**
    - Gets the message that was not filtered
  - **Parameters**
    - `aMessage` - the message that fell of the other side

- *getMessage*

  `public String getMessage( )`

- *setComposeStarMessage*

  `public void setComposeStarMessage( dotNetComposeStar.runtime.message.Message aMessage )`

  - **Usage**
    - Sets the Message that was not filtered
  - **Parameters**
    - `aMessage` - the message that fell of the other side

Methods inherited from class
`dotNetComposeStar.exception.FilterModuleException`

(in C.5.4, page 85)

Methods inherited from class
`dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.5.9 Class SelectorNotFoundException

Exception thrown while interpreting a pattern when the pattern specifies a selector not existing in the type definition of the current target. (For example inner.nonExistingMethod)

Declaration

public class SelectorNotFoundException
extends dotNetComposeStar.exception.InvalidPatternExpressionException

Serializable Fields

- private String selector
  - Name of the method to which the message was directed

Constructors

- *SelectorNotFoundException*
  `public SelectorNotFoundException( )`
  - **Usage**
    * Default Constructor

- *SelectorNotFoundException*
  `public SelectorNotFoundException( java.lang.String  message )`

Methods inherited from class `dotNetComposeStar.exception.InvalidPatternExpressionException`

(in C.5.7, page 90)

Methods inherited from class `dotNetComposeStar.exception.FilterSpecificationException`

(in C.5.5, page 86)

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`

- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement [] )`

- *toString*
  `public String toString( )`

C.5.10 Class TargetNotFoundException

Exception thrown when a pattern calls for a target - selector pair not existing within the enclosing filter module at the time of interpretation (for example *.messageNotImplementedByInnerInternalOrExternal)

Declaration

public class TargetNotFoundException
extends dotNetComposeStar.exception.InvalidPatternExpressionException

Serializable Fields

- private String target
  - Name of the internal, external or pseudo variable that was not found

Constructors

- *TargetNotFoundException*
  `public TargetNotFoundException( )`
  - **Usage**
    * Default constructor

- *TargetNotFoundException*
  `public TargetNotFoundException( java.lang.String message )`

Methods inherited from class `dotNetComposeStar.exception.InvalidPatternExpressionException`

(in C.5.7, page 90)

Methods inherited from class `dotNetComposeStar.exception.FilterSpecificationException`

(in C.5.5, page 86)

Methods inherited from class `dotNetComposeStar.exception.ComposeStarException`

(in C.5.1, page 79)

Methods inherited from class `java.lang.RuntimeException`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable fillInStackTrace( )`
- *getCause*
  `public Throwable getCause( )`
- *getLocalizedMessage*
  `public String getLocalizedMessage( )`
- *getMessage*
  `public String getMessage( )`
- *getStackTrace*
  `public StackTraceElement getStackTrace( )`
- *initCause*
  `public synchronized Throwable initCause( java.lang.Throwable   )`
- *printStackTrace*
  `public void printStackTrace( )`

- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream   )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter   )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement []  )`
- *toString*
  `public String toString( )`

C.6 Package dotNetComposeStar.runtime.message

Package Contents *Page*

C.7 Classes

C.7.1 Class Message

Models the Message as it is being Filtered

Keeps the name and arguments of the message. It also keeps some of the pseudo variables necessary during filtering (for example internals and externals). Finally it is responsible for producing a Reified message of itself for the ACT's in the meta filter.

Declaration

public class Message
extends java.lang.Object

Constructors

- *Message*
  `public Message( java.lang.String  selector )`
  - **Usage**
    * Constructs a Message from its selector

      It is supposed that no arguments are set.
  - **Parameters**
    * `selector` - selector of the message

- *Message*
  `public Message( java.lang.String  selector, java.lang.Object [] args )`
  - **Usage**
    * Constructs a message with arguments
  - **Parameters**

    - `selector` - selector of the message
    - `args` - arguments of the message

Methods

- *addFilterParameter*

  `public void addFilterParameter( java.lang.String  messageElement, java.lang.String  identifier )`

  - **Usage**
    - Sets a new filter parameter
  - **Parameters**
    - `messageElement` - the name of the parameter
    - `identifier` - the value of the parameter.

- *externals*

  `public Dictionary externals( )`

  - **Usage**
    - Returns a dictionary containing all the externals
  - **Returns** - the current externals

- *externals*

  `public void externals( dotNetComposeStar.util.Dictionary externals )`

  - **Usage**
    - Replaces all the externals.
  - **Parameters**
    - `externals` - new set of externals.

- *getArguments*

  `public Object getArguments( )`

  - **Usage**
    - Returns the arguments of this message

- **Returns** - an array of objects containing the arguments

- *getExternals*

  `public Object getExternals( java.lang.String  name )`

  - **Usage**
    * Returns an external by its name
  - **Parameters**
    * `name` - name of the desired external
  - **Returns** - the object bound to the name.

- *getFilterParameter*

  `public String getFilterParameter( java.lang.String messageElement )`

  - **Usage**
    * Returns the value associated with the message element.
  - **Parameters**
    * `messageElement` - key
  - **Returns** - the identifier associated with it

- *getInner*

  `public Object getInner( )`

  - **Usage**
    * Returns the object bound to inner

- *getInternal*

  `public Object getInternal( java.lang.String  name )`

  - **Usage**
    * Returns an internal by its name.
  - **Parameters**
    * `name` - name of the internal
  - **Returns** - the object bound to that name.

- *getSelector*

  `public String getSelector( )`

  - **Usage**
    - Returns the selector of this message
  - **Returns** - current selector

- *internals*

  `public Dictionary internals( )`

  - **Usage**
    - Returns a dictionary containing all the internals.
  - **Returns** - all the internals

- *internals*

  `public void internals( dotNetComposeStar.util.Dictionary internals )`

  - **Usage**
    - Sets (replaces) the internals of the message
  - **Parameters**
    - `internals` - new set of internals

- *reify*

  `public ReifiedMessage reify( )`

  - **Usage**
    - Reifies this message.

      It produces a version of this message that is fit for manipulation inside a ACT.
  - **Returns** - Reified version of this message

- *setInner*

  `public void setInner( java.lang.Object  inner )`

  - **Usage**
    * Binds the object to the inner pseudo var.
  - **Parameters**
    * `inner` - new inner

C.7.2 Class ReifiedMessage

Models a Reified message

It is handled by ACT methods called when a Meta Filter accepts a message. The reified message can offer information about the message, and fire, reply or send itself.

Declaration

public class ReifiedMessage
extends java.lang.Object

Fields

- public static int FIRED
  - State of the reified message - message was fired
- public static int REPLIED
  - State of the reified message - message was replied
- public static int SENT
  - State of the reified message - message was sent

Methods

- *fire*
  `public void fire( )`
  - **Usage**

    * Fires this message

- *getArgs*

  `public Object getArgs( )`

  - **Usage**
    * Returns the arguments of the message
  - **Returns** - an array containing the arguments of the message

---

- *getCallBack*

  `public ACTcallBackMethod getCallBack( )`

  - **Usage**
    * Returns the callback

      This method is only valid if the state of the ReifiedMessage is SENT.
  - **Returns** - the act callback method, if the state is SENT, null otherwise

---

- *getReturnValue*

  `public Object getReturnValue( )`

  - **Usage**
    * Returns the return value of the message

      Only valid if the state of the ReifiedMessage is SENT
  - **Returns** - the return value of the message if state is SENT, null otherwise.

- *getSelector*

  `public String getSelector( )`

  - **Usage**
    * Returns the selector of this message
  - **Returns** - the selector

- *getState*

  `public int getState( )`

  - **Usage**
    - Returns the current state of the ReifiedMessage
  - **Returns** - ReifiedMessage.FIRED, ReifiedMessage.REPLIED or ReifiedMessage.SENT

- *reply*

  `public void reply( java.lang.Object  content )`

  - **Usage**
    - Replies this message with a given object

      If the original method had a void return type, the content parameter is ignored
  - **Parameters**
    - `content` - the object to return to the original sender

- *send*

  `public void send( ACTcallBackMethod  actCallback )`

  - **Usage**
    - Sends this message.

      The actCallback is invoked on the return trip of the message once the return object is sent.
  - **Parameters**
    - `actCallback` - method to invoke

- *setSelector*

  `public void setSelector( java.lang.String  s )`

  - **Usage**
    - Sets a new selector for this message

- **Parameters**
  - `s` - the new selector

C.7.3 Class ReplyMessage

Models the reified message that is handed over to the ACTcallbacks

It allows for the replacement of the return value of the message

Declaration

public class ReplyMessage
extends dotNetComposeStar.runtime.message.ReifiedMessage

Constructors

- *ReplyMessage*
  `public ReplyMessage( dotNetComposeStar.runtime.message.Message m, java.lang.Object  returnValue )`
  - **Usage**
    - Constructs a new ReplyMessage out of a message and its return value
  - **Parameters**
    - `m` - message to reify
    - `returnValue` - return value of the message

Methods

- *setReturnObject*
  `public void setReturnObject( java.lang.Object  o )`
  - **Usage**

    * Replaces the return value with a new one.

      No checks are done to see if the new return value fills the interface requirements of the message (that is, if it is a valid type)

  - **Parameters**
    * `o` - new return value

Methods inherited from class
`dotNetComposeStar.runtime.message.ReifiedMessage`

(in C.7.2, page 105)

- *fire*
  `public void fire( )`
  - **Usage**
    * Fires this message

- *getArgs*
  `public Object getArgs( )`
  - **Usage**
    * Returns the arguments of the message
  - **Returns** - an array containing the arguments of the message

---

- *getCallBack*
  `public ACTcallBackMethod getCallBack( )`
  - **Usage**
    * Returns the callback

      This method is only valid if the state of the ReifiedMessage is SENT.
  - **Returns** - the act callback method, if the state is SENT, null otherwise

---

- *getReturnValue*
  `public Object getReturnValue( )`
  - **Usage**
    * Returns the return value of the message

      Only valid if the state of the ReifiedMessage is SENT
  - **Returns** - the return value of the message if state is SENT, null otherwise.

- *getSelector*
  `public String getSelector( )`
  - **Usage**

    * Returns the selector of this message
  - **Returns** - the selector

- *getState*
  `public int getState( )`
  - **Usage**
    * Returns the current state of the ReifiedMessage
  - **Returns** - ReifiedMessage.FIRED, ReifiedMessage.REPLIED or ReifiedMessage.SENT

- *reply*
  `public void reply( java.lang.Object  content )`
  - **Usage**
    * Replies this message with a given object

      If the original method had a void return type, the content parameter is ignored
  - **Parameters**
    * `content` - the object to return to the original sender

- *send*
  `public void send( ACTcallBackMethod  actCallback )`
  - **Usage**
    * Sends this message.

      The actCallback is invoked on the return trip of the message once the return object is sent.
  - **Parameters**
    * `actCallback` - method to invoke

- *setSelector*
  `public void setSelector( java.lang.String  s )`
  - **Usage**
    * Sets a new selector for this message
  - **Parameters**
    * `s` - the new selector

C.8 Package dotNetComposeStar.runtime

Package Contents *Page*

C.9 Classes

C.9.1 Class ComposeStarObject

Summary description for ComposeStarObject.

Base class of all objects that are to be superimposed. This class must contain the union of all the interfaces of the classes that are to be superimposed in order for the polymorphism to work.

Declaration

public class ComposeStarObject
extends ContextBoundObject

Constructors

- *ComposeStarObject*
  `public ComposeStarObject( )`

Methods

- *bar*
  `public void bar( )`
- *getCallLog*
  `public String getCallLog( )`
- *getKey*
  `public String getKey( )`
  - **Usage**
    * returns a string that identifies each object uniquely.

      The string is derived from the hashcode of the object.

  – **Returns** - a unique string for this object.

- *getLastReturn*

  `public Object getLastReturn( )`

- *handleMouse*

  `public String handleMouse( )`

- *isLocked*

  `public boolean isLocked( )`

- *lock*

  `public void lock( )`

- *one*

  `public int one( )`

- *setState*

  `public void setState( int  s )`

- *unlock*

  `public void unlock( )`

Methods inherited from class `ContextBoundObject`

C.9.2 Class Dispatch

Dispatch Filter

This filter redirects messages that accepts to the objects (internals or externals) defined in its specification.

Declaration

---

public class Dispatch
extends dotNetComposeStar.runtime.Filter

Constructors

- *Dispatch*

  `public Dispatch( java.lang.String  name,`
  `dotNetComposeStar.runtime.interpreter.FilterSpecificationInterpreter`
  `interpreter,`
  `dotNetComposeStar.runtime.interpreter.ConditionResolver  resolver`
  `)`

  – **Usage**
    * Constructs a Dispatch filter
  – **Parameters**
    * `name` - name of the filter.
    * `interpreter` - Specification of the filter (conditions and parameters)
    * `resolver` - Condition resolver for this filter.

Methods

- *acceptAction*

  `public ComposeStarAction acceptAction(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  – **Usage**
    * Accept action

      The accept action is a DispatchAction
  – **Parameters**
    * `m` - Message accepted
    * `context` - in which it was accepted
  – **Returns** - Action to dispatch the message
  – **See Also**
    * `dotNetComposeStar.runtime.actions.DispatchAction` (in C.11.3, page 148)

- *rejectAction*

  `public ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - Reject action

      The reject action is a ContinueToNextFiliterAction
  - **Parameters**
    - `m` - message rejected
    - `context` - in which it was rejected
  - **Returns** - action to continue to the next filter

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*

  `public abstract ComposeStarAction acceptAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*

  `public boolean canAccept( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - desides if the filter will accept or reject the message
  - **Parameters**
    - `m` - the message to accept
    - `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

---

- *getConditionResolver*

  `public ConditionResolver getConditionResolver( )`

  - **Usage**
    - Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

---

- *getName*
  `public String getName( )`
  - **Usage**
    - Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage( dotNetComposeStar.runtime.message.Message  m )`
  - **Usage**
    - Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**
    - `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  `public abstract ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message  m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    - Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

C.9.3 Class ErrorFilter

Model the error filter

This filter will create an ErrorAction when rejecting a message. When executed, the error action is to throw an exception

Declaration

public class ErrorFilter
extends dotNetComposeStar.runtime.Filter

Constructors

- *ErrorFilter*

  `public ErrorFilter( java.lang.String  name,`
  `dotNetComposeStar.runtime.interpreter.FilterSpecificationInterpreter`
  `interpreter,`
  `dotNetComposeStar.runtime.interpreter.ConditionResolver  resolver`
  `)`

  – **Usage**
    * Constructs a new Error filter
  – **Parameters**
    * `name` - name of the filter.
    * `interpreter` - Specification of the filter (conditions and parameters)
    * `resolver` - Condition resolver for this filter.

Methods

- *acceptAction*

  `public ComposeStarAction acceptAction(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  – **Usage**
    * Accepts the message

      Continues to the next filter
  – **Parameters**
    * `m` - Message accepted
    * `context` - in which it was accepted
  – **Returns** - action to continue to the next filter

- *rejectAction*

  `public ComposeStarAction rejectAction(`

`dotNetComposeStar.runtime.message.Message m,`
`dotNetComposeStar.util.Dictionary context )`

- **Usage**
  - ∗ Rejects the message.
- **Parameters**
  - ∗ `m` - Message rejected
  - ∗ `context` - in which it was rejected
- **Returns** - action to throw an exception

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*
  `public abstract ComposeStarAction acceptAction(`
  `dotNetComposeStar.runtime.message.Message m,`
  `dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    - ∗ Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    - ∗ `m` - Message being filtered
    - ∗ `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*
  `public boolean canAccept( dotNetComposeStar.runtime.message.Message`
  `m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    - ∗ desides if the filter will accept or reject the message
  - **Parameters**
    - ∗ `m` - the message to accept
    - ∗ `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    - ∗ Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

- *getName*
  `public String getName( )`

- **Usage**
  - Name of the filter
- **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage( dotNetComposeStar.runtime.message.Message m )`

  - **Usage**
    - Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**
    - `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  `public abstract ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

  - **Usage**
    - Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

C.9.4 Class Filter

Models all the filters in the system.

Handles the naming, keeps the condition resolver, and interprets the filter specification to decide if a filter accepts or rejects a message.

Declaration

public abstract class Filter
extends java.lang.Object

Constructors

- *Filter*

  `protected Filter( )`

  - **Usage**
    - Constructor of Filters

- *Filter*

  ```
  public Filter( java.lang.String  name,
  dotNetComposeStar.runtime.interpreter.FilterSpecificationInterpreter
  interpreter,
  dotNetComposeStar.runtime.interpreter.ConditionResolver  resolver
  )
  ```

  - **Usage**
    - Constructs out of a name, a specification and a condition resolver
  - **Parameters**
    - `name` - the name of the filter
    - `interpreter` - Specification of the filter
    - `resolver` - condition resolver

Methods

- *acceptAction*

  ```
  public abstract ComposeStarAction acceptAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    - Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation

- **Returns** - an action

- *canAccept*

  `public boolean canAccept(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    - desides if the filter will accept or reject the message
  - **Parameters**
    - `m` - the message to accept
    - `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

---

- *getConditionResolver*

  `public ConditionResolver getConditionResolver( )`

  - **Usage**
    - Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

---

- *getName*

  `public String getName( )`

  - **Usage**
    - Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*

  `public ComposeStarAction handleMessage(`
  `dotNetComposeStar.runtime.message.Message  m )`

  - **Usage**
    - Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**

    - `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*

  `public abstract ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message  m, dotNetComposeStar.util.Dictionary  context )`

  - **Usage**
    - Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

C.9.5 Class FilterModule

Models the Filter modules. Filter modules are a collection of inputfilters, conditions, methods, internals and externals.

Declaration

public abstract class FilterModule
extends java.lang.Object
implements dotNetComposeStar.runtime.interpreter.ConditionResolver

Constructors

- *FilterModule*

  `public FilterModule( java.lang.String  name )`

  - **Usage**
    - Constructs a filtermodule by a given name.

- **Parameters**
  - `name` - name of the filtermodule

Methods

- *addCondition*

  `public void addCondition( java.lang.String  name )`

  - **Usage**
    - Defines a new condition in the filter module

- *addExternal*

  `public void addExternal( java.lang.String  name, java.lang.Object value )`

  - **Usage**
    - Defines a new external in the filter module
  - **Parameters**
    - `name` - of the external
    - `value` - object bound to the name.

- *addInputFilter*

  `public void addInputFilter( dotNetComposeStar.runtime.Filter  in )`

  - **Usage**
    - Adds a new input filter

      This is added ON TOP of existing filters
  - **Parameters**
    - `in` - inputfilter

- *addInternal*

  `public void addInternal( java.lang.String  name, java.lang.Object value )`

  - **Usage**
    - Defines a new internal in the filtermodule
  - **Parameters**
    - `name` - name of the internal
    - `value` - object bound to the name.

- *addMethod*

  `public void addMethod( java.lang.String  selector )`

  - **Usage**
    - Defines a new method in the filtermodule
  - **Parameters**
    - `selector` - name of the new method

- *getExternals*

  `public Dictionary getExternals( )`

  - **Usage**
    - Gets the externals defined in the filtermodule
  - **Returns** - a dictionary containing the externals

- *getFilters*

  `public List getFilters( )`

  - **Usage**
    - Gets a list with the inputfilters of the module
  - **Returns** - inputfilters

- *getInputFilters*

  `public List getInputFilters( )`

  - **Usage**
    - Returns the input filters
  - **Returns** - a list containing the inputfilters

- *getInternals*

  `public Dictionary getInternals( )`

  - **Usage**
    - Gets the internals defined in the filtermodule
  - **Returns** - a dictionary containing the internals

- *getName*

  `public String getName( )`

  - **Usage**
    - gets the name of the filter module
  - **Returns** - a string with the name of the module

- *handleInputMessage*

  `public List handleInputMessage(`
  `dotNetComposeStar.runtime.message.Message  aMessage )`

  - **Usage**
    - Handles a message that is on the way IN the filter.

      The FilterInterface takes care of passing the message through each of its inputFilters. If the message is not filtered by any of the filters, it returns false; it return true otherwise.

      If the message is meant for one of the declared methods of the filter, it is dispatched, and true is returned.

      This is now handled by the policy package
  - **Parameters**
    - `aMessage` - the message to handle.
  - **Returns** - true if the message is handled by any of the filters, false otherwise.

- *handleOutputMessage*
  `public boolean handleOutputMessage( dotNetComposeStar.runtime.message.Message aMessage )`
  - **Usage**
    - Not implemented due to limitations of the message interception
  - **Parameters**
    - `aMessage` -

- *removeInputFilter*
  `public void removeInputFilter( java.lang.String name )`
  - **Usage**
    - removes an inputfilter identified by its name.
  - **Parameters**
    - `name` - name of the inputfilter

- *resolve*
  `public boolean resolve( java.lang.String cond )`
  - **Usage**
    - Resolves a condition.
  - **Parameters**
    - `cond` - Name of the condition to resolve
  - **Returns** - the evaluation of the condition
  - **See Also**
    - `dotNetComposeStar.runtime.intepreter.ConditionResolver`

C.9.6 Class LockingFilterModule

Example of a concrete Filter Module.

Declaration

public class LockingFilterModule
extends dotNetComposeStar.runtime.FilterModule

Constructors

- *LockingFilterModule*

  `public LockingFilterModule( java.lang.String  name )`

Methods

- *isLocked*

  `public boolean isLocked( )`

- *lock*

  `public void lock( )`

- *unlock*

  `public void unlock( )`

Methods inherited from class `dotNetComposeStar.runtime.FilterModule`

(in C.9.5, page 122)

- *addCondition*
  `public void addCondition( java.lang.String  name )`
  - **Usage**
    * Defines a new condition in the filter module
- *addExternal*
  `public void addExternal( java.lang.String  name, java.lang.Object value )`
  - **Usage**
    * Defines a new external in the filter module
  - **Parameters**
    * `name` - of the external
    * `value` - object bound to the name.

- *addInputFilter*
  `public void addInputFilter( dotNetComposeStar.runtime.Filter  in )`
  - **Usage**
    - Adds a new input filter

      This is added ON TOP of existing filters
  - **Parameters**
    - `in` - inputfilter

- *addInternal*
  `public void addInternal( java.lang.String  name, java.lang.Object value )`
  - **Usage**
    - Defines a new internal in the filtermodule
  - **Parameters**
    - `name` - name of the internal
    - `value` - object bound to the name.

- *addMethod*
  `public void addMethod( java.lang.String  selector )`
  - **Usage**
    - Defines a new method in the filtermodule
  - **Parameters**
    - `selector` - name of the new method

- *getExternals*
  `public Dictionary getExternals( )`
  - **Usage**
    - Gets the externals defined in the filtermodule
  - **Returns** - a dictionary containing the externals

- *getFilters*
  `public List getFilters( )`
  - **Usage**
    - Gets a list with the inputfilters of the module
  - **Returns** - inputfilters

- *getInputFilters*
  `public List getInputFilters( )`
  - **Usage**
    - Returns the input filters
  - **Returns** - a list containing the inputfilters

- *getInternals*
  `public Dictionary getInternals( )`
  - **Usage**
    - Gets the internals defined in the filtermodule
  - **Returns** - a dictionary containing the internals

- *getName*
  `public String getName( )`
  - **Usage**
    - gets the name of the filter module
  - **Returns** - a string with the name of the module

- *handleInputMessage*
  `public List handleInputMessage( dotNetComposeStar.runtime.message.Message  aMessage )`
  - **Usage**
    - Handles a message that is on the way IN the filter.

      The FilterInterface takes care of passing the message through each of its inputFilters. If the message is not filtered by any of the filters, it returns false; it return true otherwise.

      If the message is meant for one of the declared methods of the filter, it is dispatched, and true is returned.

      This is now handled by the policy package
  - **Parameters**
    - `aMessage` - the message to handle.
  - **Returns** - true if the message is handled by any of the filters, false otherwise.

- *handleOutputMessage*
  `public boolean handleOutputMessage( dotNetComposeStar.runtime.message.Message  aMessage )`
  - **Usage**
    - Not implemented due to limitations of the message interception
  - **Parameters**
    - `aMessage` -

- *removeInputFilter*
  `public void removeInputFilter( java.lang.String  name )`
  - **Usage**
    - removes an inputfilter identified by its name.
  - **Parameters**
    - `name` - name of the inputfilter

- *resolve*
  `public boolean resolve( java.lang.String  cond )`
  - **Usage**
    - Resolves a condition.
  - **Parameters**
    - `cond` - Name of the condition to resolve
  - **Returns** - the evaluation of the condition
  - **See Also**
    - `dotNetComposeStar.runtime.intepreter.ConditionResolver`

C.9.7 Class Meta

Models the Meta filter

If a message is accepted, it is reified and offered to a method defined in the filter specification. if the message is rejected, it is passed on to the next filter.

Declaration

public class Meta
extends dotNetComposeStar.runtime.Filter

Constructors

- *Meta*
  ```
  public Meta( java.lang.String  name,
  dotNetComposeStar.runtime.interpreter.FilterSpecificationInterpreter
  interpreter,
  dotNetComposeStar.runtime.interpreter.ConditionResolver  resolver
  )
  ```
  - **Usage**
    - Constructs a new meta filter
  - **Parameters**
    - `name` - name of the filter.

    * `interpreter` - Specification of the filter (conditions and parameters)
    * `resolver` - Condition resolver for this filter.

Methods

- *acceptAction*

  ```
  public ComposeStarAction acceptAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    * Generates the appropriate action when the filter accepts a message

      The action returned is a MetaAction.
  - **Parameters**
    * `m` - Message accepted
    * `context` - in which it was accepted
  - **Returns** - action to reify the message an pass it to a method.

- *rejectAction*

  ```
  public ComposeStarAction rejectAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

  - **Usage**
    * Generates the appropriate action when the filter rejects a message
  - **Parameters**
    * `m` - Message accepted
    * `context` - in which it was accepted
  - **Returns** - action to continue to the next filter

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*
  `public abstract ComposeStarAction acceptAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    - Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*
  `public boolean canAccept( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    - desides if the filter will accept or reject the message
  - **Parameters**
    - `m` - the message to accept
    - `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

---

- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    - Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

---

- *getName*
  `public String getName( )`
  - **Usage**
    - Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage( dotNetComposeStar.runtime.message.Message m )`
  - **Usage**
    - Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**
    - `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  ```
  public abstract ComposeStarAction rejectAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    - Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

C.9.8 Class ObjectManager

This class manages the filtering process for each object.

The an object's objectManager is obtained by with the static method getObjectManagerFor. Subsequent calls to the method return the original object manager; this means that it exists a single object manager for each object.

It is the responsibility of the objectManager to receive intercepted messages from the MessageInterceptionLayer. And to start the filtering process. The Object manager is NOT thread safe.

Finally, the object manager also manages the filtermodules that are imposed on the object that manages.

Declaration

public class ObjectManager
extends java.lang.Object

Fields

- public Object theObject
  - The Object that is being managed

Constructors

- *ObjectManager*

  `public ObjectManager( )`

  - **Usage**
    - Constructs a new Object Manager - made public for testing purposes.

Methods

- *addFilterModule*

  `public void addFilterModule( dotNetComposeStar.runtime.FilterModule  newModule )`

  - **Usage**
    - Adds a new filter module on top of the existing ones
  - **Parameters**
    - `newModule` - the new filter module

- *deliverMessage*

  `public Object deliverMessage( java.lang.Object  receiver, java.lang.String  selector, java.lang.Object [] args )`

  - **Usage**
    - Delivers a message to the object that is being managed.

      This is the point of contact with the Message Interception Layer. The message is filtered and a return value is given.
  - **Parameters**
    - `receiver` - of the message
    - `selector` - name of the method
    - `args` - array containing the arguments of the message
  - **Returns** - the return value of the message, if the method's return type is void return null.

- *getFilterModule*
  `public FilterModule getFilterModule( java.lang.String  moduleName )`
  - **Usage**
    * Retrieves a Filtermodule by its name.
  - **Parameters**
    * `moduleName` - the name of the desired filtermodule
  - **Returns** - the filter module if it is imposed on the managed object, null otherwise

---

- *getFilterModules*
  `public List getFilterModules( )`
  - **Usage**
    * Returns a listing of the filter modules in the order in which messages are passed
  - **Returns** - a list containing the modules

- *getObjectManagerFor*
  `public static ObjectManager getObjectManagerFor( dotNetComposeStar.runtime.ComposeStarObject  o )`
  - **Usage**
    * Retrieves the Object Manager associated with the object o. If there is none, a new one is created.
  - **Parameters**
    * `o` - The object whose manager is needed.
  - **Returns** - the manager of the object o

- *receiveMessage*
  `public Object receiveMessage( dotNetComposeStar.runtime.message.Message  aMessage )`
  - **Usage**

    * Receives a message and filters it, returning the appropriate value

      This method passes the received message through each of the filter modules attached to the managed object.

      The way the message is treated by each filter module is given by a policy

  - **Parameters**
    * `aMessage` - the message received
  - **Returns** - the return value of the message, if the method's return type is void return null
  - **See Also**
    * `dotNetComposeStar.runtime.policy.FilterPolicy` (in C.13.1, page 157)

- *removeFilterModule*
  `public void removeFilterModule( dotNetComposeStar.runtime.FilterModule  mod )`
  - **Usage**
    * Removes a module
  - **Parameters**
    * `mod` - module to remove

- *removeFilterModule*
  `public void removeFilterModule( java.lang.String  moduleName )`
  - **Usage**
    * Removes a filter module given its name
  - **Parameters**
    * `moduleName` - the name of the module

- *reset*
  `public static void reset( )`

- **Usage**
  - Resets the bindings of the objects - managers

    This method is provided for testing purposes

C.9.9 Class Send

Models the Send filter

Not implemented because it is an output filter, and those are not implemented.

Declaration

public class Send
extends dotNetComposeStar.runtime.Filter

Constructors

- *Send*
  `public Send( )`

Methods

- *acceptAction*
  ```
  public ComposeStarAction acceptAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

- *rejectAction*
  ```
  public ComposeStarAction rejectAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*
  `public abstract ComposeStarAction acceptAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    * Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    * `m` - Message being filtered
    * `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*
  `public boolean canAccept( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    * desides if the filter will accept or reject the message
  - **Parameters**
    * `m` - the message to accept
    * `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    * Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

- *getName*
  `public String getName( )`
  - **Usage**
    * Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage( dotNetComposeStar.runtime.message.Message m )`
  - **Usage**
    * Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**

    * `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  `public abstract ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`
  - **Usage**
    * Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    * `m` - Message being filtered
    * `context` - context of the interpretation
  - **Returns** - an action

C.9.10 Class Substitution

Declaration

public class Substitution
extends dotNetComposeStar.runtime.Filter

Constructors

- *Substitution*
  `public Substitution( )`

Methods

- *acceptAction*
  `public ComposeStarAction acceptAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

- *rejectAction*
  `public ComposeStarAction rejectAction( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary context )`

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*
  `public abstract ComposeStarAction acceptAction( dotNetComposeStar.runtime.message.Message  m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    * `m` - Message being filtered
    * `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*
  `public boolean canAccept( dotNetComposeStar.runtime.message.Message m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * desides if the filter will accept or reject the message
  - **Parameters**
    * `m` - the message to accept
    * `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    * Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

- *getName*
  `public String getName( )`
  - **Usage**
    * Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage( dotNetComposeStar.runtime.message.Message  m )`
  - **Usage**
    * Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**

    * `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  ```
  public abstract ComposeStarAction rejectAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```
  - **Usage**
    * Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    * `m` - Message being filtered
    * `context` - context of the interpretation
  - **Returns** - an action

C.9.11 Class Wait

Not Implemented

Declaration

public class Wait
extends dotNetComposeStar.runtime.Filter

Constructors

- *Wait*
  ```
  public Wait( )
  ```

Methods

- *acceptAction*
  ```
  public ComposeStarAction acceptAction(
  dotNetComposeStar.runtime.message.Message  m,
  dotNetComposeStar.util.Dictionary  context )
  ```

- *rejectAction*

  `public ComposeStarAction rejectAction(`

  `dotNetComposeStar.runtime.message.Message  m,`

  `dotNetComposeStar.util.Dictionary  context )`

Methods inherited from class `dotNetComposeStar.runtime.Filter`

(in C.9.4, page 119)

- *acceptAction*
  `public abstract ComposeStarAction acceptAction(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * Produces the action to carry out of the message is accepted by the filter
  - **Parameters**
    * `m` - Message being filtered
    * `context` - context of the interpretation
  - **Returns** - an action

- *canAccept*
  `public boolean canAccept( dotNetComposeStar.runtime.message.Message`
  `m, dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    * desides if the filter will accept or reject the message
  - **Parameters**
    * `m` - the message to accept
    * `context` - of the interpretation
  - **Returns** - true if the message is accepted, false otherwise

  ---
- *getConditionResolver*
  `public ConditionResolver getConditionResolver( )`
  - **Usage**
    * Returns the condition resolver
  - **Returns** - the object that is evaluating the conditions

  ---
- *getName*
  `public String getName( )`
  - **Usage**
    * Name of the filter
  - **Returns** - the name of the filter

- *handleMessage*
  `public ComposeStarAction handleMessage(`
  `dotNetComposeStar.runtime.message.Message  m )`
  - **Usage**
    - Passes the object by the specification to see if the filter will accept the message or not and returns the corresponding action
  - **Parameters**
    - `aMessage` - message being filtered
  - **Returns** - boolean action to carry out.

- *rejectAction*
  `public abstract ComposeStarAction rejectAction(`
  `dotNetComposeStar.runtime.message.Message  m,`
  `dotNetComposeStar.util.Dictionary  context )`
  - **Usage**
    - Produces the action to carry out if the message is rejected by the filter
  - **Parameters**
    - `m` - Message being filtered
    - `context` - context of the interpretation
  - **Returns** - an action

C.10 Package dotNetComposeStar.runtime.actions

Package Contents *Page*

C.11 Classes

C.11.1 Class ComposeStarAction

Parent class of all Actions

Actions dictate the result of a message passing by a filter. In particular, an action can just be to pass on to the next filter

Declaration

public abstract class ComposeStarAction
extends java.lang.Object

Constructors

- *ComposeStarAction*
  `protected ComposeStarAction( boolean  accepted )`
  - **Usage**
    * Constructs a new ComposeStarAction

      Only classes that extend ComposeStarAction can access it.
  - **Parameters**
    * `accepted` - if the filter accepted the message

Methods

- *execute*
  `public abstract Object execute( )`
  - **Usage**
    * Execute this action.

Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.

- **Returns** - the object that is to be sent to the original sender of the message, null if there is none.

- *getShouldContinue*

  `public boolean getShouldContinue( )`

  - **Usage**
    - Says if the message should continue in the filtering process or not.
  - **Returns** - TRUE if it should continue, FALSE otherwise.

- *wasAccepted*

  `public boolean wasAccepted( )`

  - **Usage**
    - Says if the message was accepted.

      Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.11.2 Class ContinueToNextFilterAction

Continues to the next filter.

This action does nothing, and allows for the continuing of the filtering of the message

Declaration

public class ContinueToNextFilterAction
extends dotNetComposeStar.runtime.actions.ComposeStarAction

Constructors

- *ContinueToNextFilterAction*

  `public ContinueToNextFilterAction( boolean  accepted )`

  - **Usage**
    - Constructs a new Continue to Next Filter Action.
  - **Parameters**
    - `accepted` - says if the message was accepted by this filter or not.

Methods

- *execute*

  `public Object execute( )`

  - **Usage**
    - Does nothing
  - **Returns** - always null

Methods inherited from class `dotNetComposeStar.runtime.actions.ComposeStarAction`

(in C.11.1, page 145)

- *execute*
  `public abstract Object execute( )`

  - **Usage**
    - Execute this action.

      Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.
  - **Returns** - the object that is to be sent to the original sender of the message, null if there is none.

- *getShouldContinue*
  `public boolean getShouldContinue( )`

  - **Usage**
    - Says if the message should continue in the filtering process or not.

  - **Returns** - TRUE if it should continue, FALSE otherwise.

- *wasAccepted*
  `public boolean wasAccepted( )`
  - **Usage**
    * Says if the message was accepted.

      Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.11.3 Class DispatchAction

Models the action to carry out when a Dispatch Filter accepts a message.

When executed, it redirects the message to the target specified during filtering of the message in the Filter Specification.

Declaration

public class DispatchAction
extends dotNetComposeStar.runtime.actions.ComposeStarAction

Constructors

- *DispatchAction*
  `public DispatchAction( boolean  accepted, java.lang.Object target, java.lang.String  selector, java.lang.Object [] args )`
  - **Usage**
    * Constructs a Dispatch Action with all the necessary information to do the invocation.
  - **Parameters**
    * `target` - the instance to which the message is to be directed
    * `selector` - method to send the message to
    * `args` - arguments of the message

Methods

- *execute*

  `public Object execute( )`

  - **Usage**
    - Dispatches the message and returns what comes from the invocation.

      This is actually delegated to the Invoker in the util package
  - **Returns** - what ever the message dispatched returned.
  - **See Also**
    - `dotNetComposeStar.util.Invoker.invoke`

Methods inherited from class `dotNetComposeStar.runtime.actions.ComposeStarAction`

(in C.11.1, page 145)

- *execute*

  `public abstract Object execute( )`

  - **Usage**
    - Execute this action.

      Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.
  - **Returns** - the object that is to be sent to the original sender of the message, null if there is none.

- *getShouldContinue*

  `public boolean getShouldContinue( )`

  - **Usage**
    - Says if the message should continue in the filtering process or not.
  - **Returns** - TRUE if it should continue, FALSE otherwise.

- *wasAccepted*

  `public boolean wasAccepted( )`

  - **Usage**
    - Says if the message was accepted.

      Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.11.4 Class ErrorAction

Models the action that is produced when a message is rejected by an Error Filter

The action defined is the throw of a ErrorFilterException.

Declaration

public class ErrorAction
extends dotNetComposeStar.runtime.actions.ComposeStarAction

Constructors

- *ErrorAction*
  ```
  public ErrorAction( boolean  accepted,
  dotNetComposeStar.exception.ErrorFilterException  exception )
  ```
  - **Usage**
    - Constructs an error action

      The error action takes the exception to be thrown, and throws it when it's executed.
  - **Parameters**
    - `accepted` - says if the filter accepted the message or not
    - `exception` - the exception to throw when the action is executed

Methods

- *execute*
  ```
  public Object execute( )
  ```
  - **Usage**
    - Executes the action by throwing an exception
  - **Returns** - always return null.

Methods inherited from class
`dotNetComposeStar.runtime.actions.ComposeStarAction`

(in C.11.1, page 145)

- *execute*
  `public abstract Object execute( )`
  - **Usage**
    - Execute this action.

      Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.
  - **Returns** - the object that is to be sent to the original sender of the message, null if there is none.
- *getShouldContinue*
  `public boolean getShouldContinue( )`
  - **Usage**
    - Says if the message should continue in the filtering process or not.
  - **Returns** - TRUE if it should continue, FALSE otherwise.
- *wasAccepted*
  `public boolean wasAccepted( )`
  - **Usage**
    - Says if the message was accepted.

      Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.11.5 Class MetaAction

Models the action that comes from the acceptance of a message by a Meta filter.

The execution of this action consists on getting the ACT method and invoking it with the reified message as a parameter. Once the ACT method returns, the reified method is checked to see if and how it was activated. Depending on this: if the message was fired, it continues its normal path; if it was Replied, the return value is saved, and the message exits the filtering process; and if it was sent, the callback is saved, and the message is allowed to move on to the next filter.

Declaration

public class MetaAction
extends dotNetComposeStar.runtime.actions.ComposeStarAction

Constructors

- *MetaAction*
  `public MetaAction( dotNetComposeStar.runtime.message.ReifiedMessage rm, java.lang.Object act, java.lang.String actSelector, boolean accepted )`
  - **Usage**
    * Constructs a Meta action.

      For this is needed a reified version of the message, a act and a method in it, and whether or not the message was accepted
  - **Parameters**
    * `rm` - the reified version of the message
    * `act` - the ACT to handle the reified message
    * `actSelector` - the name of the method that handles the reified message
    * `accepted` - says if the message was accepted or not.

Methods

- *execute*
  `public Object execute( )`
  - **Usage**
    * Executes this MetaAction

      This process is explained at the beginning of this class.

  - **Returns** - depends on the handling of the reified message by the ACT method
  - **See Also**
    - `dotNetComposeStar.runtime.actions.MetaAction` (in C.11.5, page 151)

- *getCallback*
  `public ACTcallBackMethod getCallback( )`
  - **Usage**
    - Returns the callback associated with the execution of this action.
  - **Returns** - a delegate for the ACT callback method if the reified message was sent; null otherwise

Methods inherited from class
`dotNetComposeStar.runtime.actions.ComposeStarAction`

(in C.11.1, page 145)

- *execute*
  `public abstract Object execute( )`
  - **Usage**
    - Execute this action.

      Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.
  - **Returns** - the object that is to be sent to the original sender of the message, null if there is none.

- *getShouldContinue*
  `public boolean getShouldContinue( )`
  - **Usage**
    - Says if the message should continue in the filtering process or not.
  - **Returns** - TRUE if it should continue, FALSE otherwise.

- *wasAccepted*
  `public boolean wasAccepted( )`
  - **Usage**

      * Says if the message was accepted.

        Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.11.6 Class WaitAction

Not implemented!!

Declaration

public class WaitAction
extends dotNetComposeStar.runtime.actions.ComposeStarAction

Constructors

- *WaitAction*
  ```
  public WaitAction( dotNetComposeStar.runtime.Filter
  currentFiliter, dotNetComposeStar.runtime.message.Message  m )
  ```

Methods

- *execute*
  ```
  public Object execute( )
  ```

Methods inherited from class `dotNetComposeStar.runtime.actions.ComposeStarAction`

(in C.11.1, page 145)
- *execute*
  ```
  public abstract Object execute( )
  ```
  - **Usage**
      * Execute this action.

        Does whatever the action is supposed to do. In case the action results in the production of a reply for the sender of the message, this Object is to be returned. If there is no result to return, null.

  - **Returns** - the object that is to be sent to the original sender of the message, null if there is none.

- *getShouldContinue*
  `public boolean getShouldContinue( )`
  - **Usage**
    * Says if the message should continue in the filtering process or not.
  - **Returns** - TRUE if it should continue, FALSE otherwise.

- *wasAccepted*
  `public boolean wasAccepted( )`
  - **Usage**
    * Says if the message was accepted.

      Another way to read this is to say that this action is a result of the message being accepted.
  - **Returns** - boolean TRUE if the message was accepted, False otherwise.

C.12 Package dotNetComposeStar.runtime.policy

Package Contents *Page*

C.13 Classes

C.13.1 Class FilterPolicy

Models the Way messages are filtered.

This deals with the way messages are handled within a FilterModule.

Declaration

public abstract class FilterPolicy
extends java.lang.Object

Constructors

- *FilterPolicy*

  `protected FilterPolicy( )`

  – **Usage**

    ∗ Constructs a new FilterPolicy

Methods

- *executeFilterPolicy*

  ```
  public abstract PolicyExecutionResult executeFilterPolicy(
  dotNetComposeStar.runtime.FilterModule  fm,
  dotNetComposeStar.util.List  filterList,
  dotNetComposeStar.runtime.message.Message  aMessage )
  ```

  – **Usage**

    ∗ Executes this filter policy on the message given.

  – **Parameters**

    ∗ `fm` - current FilterModule
    ∗ `filterList` - list of filter within the filtermodule
    ∗ `aMessage` - message to filter

- *getPolicy*

  `public static final FilterPolicy getPolicy( )`

  - **Usage**
    - Returns the default FilterPolicy

      Works as a factory method.
  - **Returns** - a FilterPolicy

C.13.2 Class PolicyExecutionResult

Models the Result of the execution of a FilterPolicy.

It says if the message should continue to the next filtermodule, the last action taken, and a stack of the ACT callbacks collected by the sent messages in the Meta Filters

Declaration

public class PolicyExecutionResult
extends java.lang.Object

Constructors

- *PolicyExecutionResult*

  `public PolicyExecutionResult( boolean  wasAccepted, boolean shouldContinue, java.lang.Object  actionResult )`

  - **Usage**
    - Constructs a Policy execution result without callbacks
  - **Parameters**
    - `wasAccepted` - says if the message was accepted during the filtering
    - `shouldContinue` - says if the message should continue to the next filter.

    * `actionResult` - the result of the last action taken

- *PolicyExecutionResult*

  `public PolicyExecutionResult( boolean  wasAccepted, boolean shouldContinue, java.lang.Object  actionResult, dotNetComposeStar.util.ComposestarStack  stack )`

  – **Usage**
    * Constructs a policy execution result with a stack of callbacks
  – **Parameters**
    * `wasAccepted` - says if the message was accepted during the filtering
    * `shouldContinue` - says if the message should continue to the next filter.
    * `actionResult` - the result of the last action taken
    * `stack` - stack of callbacks

Methods

- *getActionResult*

  `public Object getActionResult( )`

  – **Usage**
    * Returns the result of the last action;
  – **Returns** - the result of the last action.

- *getCallbacks*

  `public ComposestarStack getCallbacks( )`

  – **Usage**
    * Returns the callbacks that resulted from the execution
  – **Returns** - the callbacks if there were any, null otherwise

---

- *shouldContinue*

  `public boolean shouldContinue( )`

  – **Usage**

    - ∗ Tests if the message should continue
  - **Returns** - TRUE if the message should continue

- *wasAccepted*

  `public boolean wasAccepted( )`

  - **Usage**
    - ∗ Tests if the message was accepted
  - **Returns** - TRUE if the message was accepted

C.14 Package dotNetComposeStar.util

Package Contents *Page*

C.15 Classes

C.15.1 Class ComposestarStack

Summary description for Stack.

Declaration

public class ComposestarStack
extends java.lang.Object

Constructors

- *ComposestarStack*

  `public ComposestarStack( )`

Methods

- *length*

  `public int length( )`

- *pop*

  `public Object pop( )`

- *push*

  `public void push( java.lang.Object  o )`

- *pushAll*

  `public void pushAll( dotNetComposeStar.util.ComposestarStack  s )`

C.15.2 Class Dictionary

<summary>

Adapter for framework specific Dictionary. In .Net, the default dictionary implementation is System.Collections.Hashtable

</summary>

Declaration

---

public class Dictionary
extends java.lang.Object

Constructors

- *Dictionary*

  `public Dictionary( )`

Methods

- *get*

  `public Object get( java.lang.Object  key )`

- *hasKey*

  `public boolean hasKey( java.lang.Object  key )`

- *keys*

  `public List keys( )`

- *put*

  `public void put( java.lang.Object  key, java.lang.Object  value )`

- *size*

  `public int size( )`

- *values*

  `public List values( )`

C.15.3 Class FilterSpecFactory

Summary description for FilterSpecFactory.

Declaration

public class FilterSpecFactory
extends java.lang.Object

Constructors

- *FilterSpecFactory*

  `public FilterSpecFactory( java.lang.String  path )`

Methods

- *process*

  `protected Object process( java.lang.Object  n )`

C.15.4 Class Invoker

This class exists to concentrate the platform dependant calls to invoke methods

Declaration

public class Invoker
extends java.lang.Object

Constructors

- *Invoker*

  `public Invoker( )`

Methods

- *invoke*

  `public static Object invoke( java.lang.Object  target, java.lang.String  selector, java.lang.Object [] args )`

C.15.5 Class List

Adapter for framework specific Dictionary. In .Net, the default dictionary implementation is System.Collections.Hashtable

Declaration

public class List
extends java.lang.Object

Constructors

- *List*

  `public List( )`

Methods

- *add*

  `public void add( java.lang.Object  o )`

- *addAll*

  `public void addAll( dotNetComposeStar.util.List  l )`

- *addFirst*

  `public void addFirst( java.lang.Object  o )`

- *contains*

  `public boolean contains( java.lang.Object  o )`

- *elementAt*

  `public Object elementAt( int  index )`

- *indexOf*

  `public int indexOf( java.lang.Object  o )`

- *remove*

  `public void remove( java.lang.Object  o )`

- *size*

  `public int size( )`

C.15.6 Class Queue

Summary description for Queue.

Declaration

public class Queue
extends java.lang.Object

Constructors

- *Queue*

  `public Queue( )`

Methods

- *pop*

  `public Object pop( )`

- *push*

  `public void push( java.lang.Object  o )`

- *size*

  `public int size( )`

- *top*

  `public Object top( )`