Object Oriented Language Interoperability

A Case Study of BETA support in Eclipse

Master's Thesis by Mads Brøgger Enevoldsen

Department of Computer Science, University of Aarhus
May 2004

Abstract

This thesis evaluates different state of the art technologies that supports language interoperability. The evaluation is done from the point of view of the programmer – it should be as convenient and easy to use many languages that interoperates together as it is to use just one language.

The second part of the thesis is a case study of the Java based Eclipse platform. Extensions to Eclipse consists of Java class files, which means that Eclipse expects plug-ins to have some relation to the Java language. A key issue is language interoperability, specifically when developing Eclipse support for languages other than Java.

Integration of the MjolnerTool (implemented in BETA) into Eclipse has been tried with two different strategies. The first using JNI to access C and thereby using C to bridge between Java and BETA. The solution works but is complex and tiresome to implement. The second strategy uses a modified BETA compiler that generates Java bytecode. This allows for a much easier integration into the Eclipse framework. The Eclipse Java debugger is reused to debug BETA source code.

Acknowledgements

I would like to thank the following people: Peter Andersen, Karsten Strandgaard Jørgensen, Mikkel Ricky Christensen, and Lasse Pedersen.

Contents

1 Introduction

This thesis is about language interoperability, more specifically interoperability among object oriented languages. Is there a need for language interoperability, one could ask. Everything can be expressed in most modern languages, so why work with more than one language? The answers are many. First of all, there is the reuse part. If the programmer has to implement specific features in his program why bother if this functionality is already implemented in another language? Since there has always been many different programming languages why not allow them to interoperate – this would certainly ease the programmer's job. Second, the use of frameworks does not allow for a free choice of programming language. The framework dictates how extensions should be written. Language interoperability could perhaps allow programmers to implement their extensions in whatever language desired.

The thesis discusses different approaches to language interoperability and uses these approaches to create support for the BETA language in the general IDE platform Eclipse. The focus is practical in the sense that it is not interesting to make it conceptually work it has to be practical seen from the programmer's point of view – interoperating with other languages should not be a pain. Extending a class from other languages should be expressed as if it is an extension of a class implemented in the language he is programming in.

Since the word BETA occurs in the subtitle of this thesis let us say it once and for all: knowledge of the programming language BETA is assumed. For information on BETA see [13].

1.1 Structure

This thesis is logically divided into two parts. The first part concerns state of the art products in support for language interoperability. The second part is a case study of the suitability of the Eclipse platform to support an IDE for the BETA language – as hinted in the subtitle of this thesis. Overall the structure is as follows:

- As an introduction to the first part chapter 2, called Language Interoperability Definitions, defines what *Object Oriented Language Interoperability* means in this thesis and states the demands found necessary to allow for programmer friendly interoperability between languages. These demands will be used as reference when evaluating the different products supporting language interoperability.

- Chapters 3 to 5 is the first logical part of the thesis. They describe and evaluate different architectures supporting language interoperability. Chapter 3 is about the OMG standard, CORBA. CORBA supports language interoperability by means of method invocation on distributed objects which can be implemented in different languages. Chapter 4 describes the Microsoft COM standard – a binary standard for expressing components. Chapter 5 deals with two different execution environments, JVM and CLR. Despite the differences these two products are conceptually the same, and they support language interoperability by providing standardised run time environments.

- Chapters 6 to 8 constitute the second part. Chapter 6 is a description of the Eclipse platform with focus on how to extend it, i.e. write plug-ins. Chapter 7 is a description of how the support for BETA is implemented in Eclipse, utilising different language interoperability approaches. Chapter 8 describes how debugging BETA source code is supported in Eclipse.

- Chapter 9 closes the second part with comments on the Eclipse architecture as a plug-in development platform and the problems encountered during development of the BETA IDE.

1.2 Terminology

IDE Integrated Development Environment – a tool supporting programming in a specific language.

API Application Programming Interface – Defines how to access a software-based service. An API is a published specification that describes how other software programs can access the functions of an automated service.

SDK Software Development Kit – a software package with documentation and tools to build applications.

CVS Concurrent Versions System – a program allowing for multiple user development on a project.

Marshalling/Unmarshalling Marshalling is the process of talking a collection of data items and assembling them in a form suitable for network transmission. Unmarshalling is the reverse process.

2 Language Interoperability Definitions

According to the Merriam-Webster Online dictionary[1] interoperability is defined to be:

> *"ability of a system (as a weapons system) to use the parts or equipment of another system"*

In terms of computer programming languages interoperability is called *language interoperability*, and therefore means that one programming language, *somehow* uses *something* written in another language. Many different programming languages exist, so if a programming language supports language interoperability in a seamless way program development using that language is easier. If the support for language interoperability is clumsy or not present at all it makes program development harder. The problem of allowing languages to interoperate in an easy and painless way is therefore an interesting problem.

The definition of language interoperability derived from the dictionary is very broad. One language *somehow* uses *something* written in another. To be able to evaluate the different products supporting language interoperability in programming languages a more precise definition is needed.

Since language interoperability is an interesting problem many different solutions have been implemented. This thesis focuses on interoperability among object oriented languages seen from the programmer's point of view. The following demands state the author's suggestion to what language interoperability *ideally* is.

1. **First Class Citizens**: Objects and classes from other languages should be considered first class citizens. I.e. there should be no distinction between objects and classes written in the source language and objects and classes written in other. There are, of course, many other concepts in object oriented programming languages than objects and classes, but this demand is to underline that we want more than just cross language method invocation – this demand will allow for cross language inheritance, which is particularly interesting.

[1] http://www.m-w.com

2. **One Language, One Compiler**: The use of objects and classes from other languages is expressed in the source language (this follows from 1), and nothing more is needed – i.e. you do not have to write anything in a middle language to tie the pieces together. The programmer is able to write the code using only one language. In addition to that, the code – interoperating with different languages – has to be compilable with the use of only one compiler. Classes from other languages do not have to be compiled with a special compiler to make it work – one compiler on the source language only.

3. **Type Safety**: The usage of objects and classes from other languages is type safe. I.e. at compile time the compiler can check and ensure that e.g. method arguments agree with the declared signatures. What about languages that are not statically typed? The meaning is that interoperability among statically typed languages (such as Java and BETA) should also be statically typed.

3 CORBA

The Object Management Group[1] (OMG) is a consortium created in 1989 with the purpose of promoting theory and practice of object technology in distributed computing systems. OMG does not produce software – only specifications.

One of their specifications is the Common Object Request Broker Architecture (CORBA). CORBA is a specification of a system that provides interoperability between objects in heterogeneous, distributed environments designed on the OMG Object Model, which is another of OMG's specifications.

3.1 OMG Object Model

The OMG Object Model (OOM) can briefly be described by the quote shown below, taken from [6]:

> *"An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulation interface."*

Invocations from clients to providers of services (from now on called *object implementations*) take place through commonly known interfaces. Figure 3.1 shows the logical view of the OOM.

Types

OOM supports the following types:

Basic types : The usual basic types available in programming languages, like: `long`, `float`, `double`, `char`, `boolean`, etc.

Constructed types : These types are `struct`, `sequence`, `union` and `array`.

[1] http://www.omg.org/

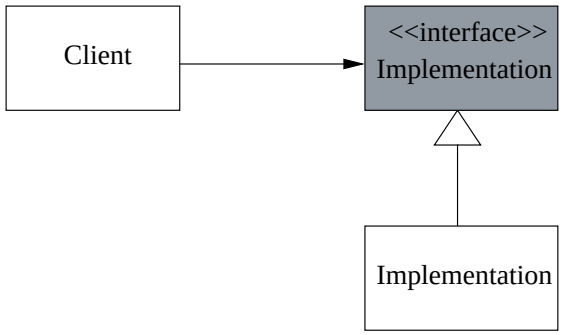

Figure 3.1: Objects interoperate through common interfaces

Interface Definition Language

Interfaces in OOM must be declared in another of OMGs standards, namely OMG IDL (Interface Definition Language). OMG IDL is a declarative language. It is used to express interfaces i.e. tells clients what operations are available, and tells object implementations what methods to implement. The general signature of an interface operation or method is the following:

```
[oneway]<return type><method name>(arg1,arg2,...,argN)
        [raises(excep1,excep2,...,excepM)]
```

OOM supports two different call semantics:

- At-most-once: If a method successfully returns, the semantics is exactly-once. But since exceptions could be thrown the semantics can only be at-most-once.
- With the optional keyword `oneway` on declarations of methods that has no return value best-effort semantics is used, meaning that the request is carried out and if either it succeeds or fails the caller will not know.

Each parameter in a function must be preceded with `in`, `out` or `inout`, meaning that the parameter is passed as either an input, output or both.

An example of an IDL interface is shown below.

```
interface Util {

  void setValue(
      in short n,
      in short m,
      in string value);

  string getValue(
      in short n,
      in short m);

  void setName(in string name);
};
```

IDL allows inheritance between interfaces. The syntax is `interface A:B`, meaning that interface A inherits from interface B. IDL also allows multiple interface inheritance supported in the following syntax: `interface A: B,C,D`.

3.2 CORBA

CORBA is a distributed interoperability architecture and the model of method invocation closely resembles a general RMI (Remote Method Invocation) model as can be found in [5]. Method invocation in CORBA is shown in figure 3.2. The typical components of an RMI system includes a remote reference module (containing the unique mapping from remote object references to actual object implementations) and a skeleton object (handling the unmarshalling of the arguments and dispatch to the correct method in the object implementation). In CORBA both these elements are encapsulated in the so-called Object Request Broker (ORB), which also acts as the communication module.

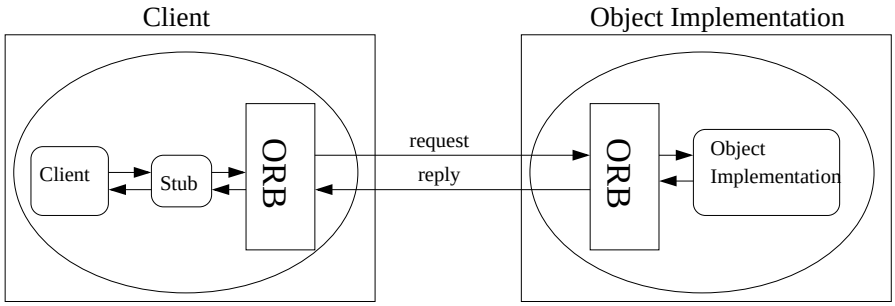

Figure 3.2: The CORBA RMI model

3.3 The Object Request Broker

The Object Request Broker (ORB) defined in the CORBA standard has many responsibilities which among others include: in response to a method invocation request, locating the object implementation, activating the object if necessary, delegating the call with arguments and returning the result to the client. The CORBA standard defines what services an ORB must provide and how the interface to these services looks. The implementation details, such as whether the ORB is in one thread or is split into components, is completely left to the ORB implementors.

Before going into detail with the parts of the ORB we have to mention two additional elements in the CORBA architecture: the interface repository and the implementation repository. Both repositories can only be accessed by the ORB. The purpose of these is to give the clients and servers a dynamic way of adding interfaces and implementations in their applications.

The ORB consists of many parts. In figure 3.3 the internal structure of the ORB is shown. The ORB core handles all the low level details of the ORB, such as the internal representation of objects and the actual communication of requests. As can be seen in the figure, clients and object implementations access the ORB through certain interfaces, each described below.

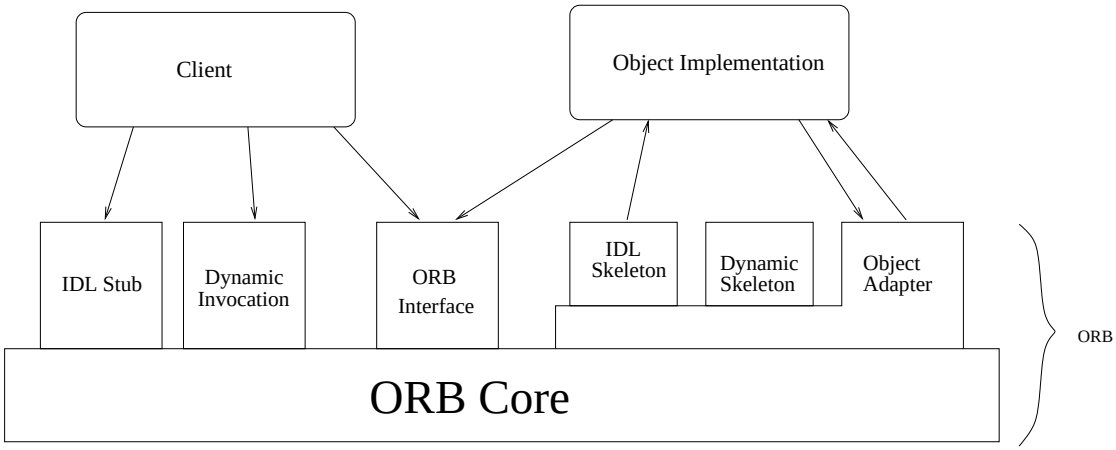

Figure 3.3: A closer look at the ORB

IDL Stub

An IDL stub contains the same method signatures as declared in the IDL file. The clients has an object reference for the remote object, and when a method is invoked

on the references the call is made to the stub. The stub then calls down into the ORB core using private interfaces known only by the stub and dependent on the actual implementation of the ORB core. The stub is the mapping between the client language to the ORB core. The client can be written in any language as long as the ORB can handle that language mapping[2].

Dynamic Invocation

Instead of using a stub when calling a specific method on an object the client can use the dynamic invocation interface. This interface is used when no stub is available to the client. As explained above, stubs are created from an IDL file, but if a server is already running new methods can be added without having to recompile and restart the server and client. The client can query the dynamic interface repository (this is done via the ORB interface) that lists the method signatures of the dynamically added interfaces. To perform the call through the dynamic invocation interface the client has to specify the object to be invoked, what operation that has to be performed, and the arguments for the call.

Object Adaptor

In figure 3.3 you see that the object adapter lies underneath the skeleton and above the ORB core. The Object Adapter is the interface that the object implementation uses to access the ORB. The Object Adapter usually has the responsibilities of generating and interpreting object references, activating objects, activating implementations, invoking methods, deactivating objects, deactivating implementations, mapping references corresponding to object implementations, and registration of implementations.

Objects integrated with the CORBA framework can be very different in Nature, and therefore it is impossible for the ORB core to specify a single interface that is efficient and easily usable to all object implementations. It is up to the implementors of the ORB to decide what object adapters to provide, and these object adapters tailor the ORB to be convenient for some types of objects. Consider an ORB communicating with a Java object in contrast to an ORB communicating with a database. Clearly there are huge differences in the communication with the two systems.

The Portable Object Adapter (POA) is an OMG standard object adapter designed to satisfy a wide variety of object implementations.

[2]Language mappings are defined by OMG for e.g. C++ and Java

IDL Skeleton

The IDL skeleton is the server side analogue to the IDL stub. The skeleton is created by the IDL compiler on the server side and contains the same method signatures as declared in the IDL file. When called the skeleton delegates the call to the actual object implementation. The call comes from the ORB and passes through the object adapter.

Dynamic Skeleton

On the client side there is the Dynamic Invocation Interface that allows clients to call object implementation not known at compile time. A similar interface is available on the server side, namely the Dynamic Skeleton Interface – this interface allows a server to call object implementation not available at compile time. The interface repository can again be used to fetch type information about the implementations corresponding to IDL interface.

ORB Interface

The ORB interface provides access to the ORB core and contains methods that are useful to both clients and object implementations. The ORB interface is defined by OMG and is therefore independent of what object adapter is used.

Example: Java to Java

To conclude this section we will take a look at a concrete example. This example will give an idea of how a programming language uses CORBA to implement an RMI model.

This example is Java to Java and is used to illustrate how a Java implementation of CORBA chooses to structure the classes of the client and object implementation. The IDL interface used in this example can be seen below.

```
interface Hello {
   string say_hello(in string name);
};
```

We want to implement the client and server in Java, so first of all we need an ORB implemented in Java. Fortunately the `JDK` library has a package `org.omg.CORBA` which among other things contains an ORB. An "IDL to Java" compiler, called `idlj`, is also

available in the JDK library. This compiler can produce Java code to be used on the client side as well as on the server side.

What is left for the application programmer is to implement a client and a server containing an actual implementation of the method defined in the IDL file. Compiling the IDL file produces some files that should be used by the client and server in a way shown in figure 3.4. The gray classes are created by the IDL compiler, and the white classes are written by the application programmer. In fact the `ildj` compiler produces some additional helper classes which are omitted here to avoid confusion.

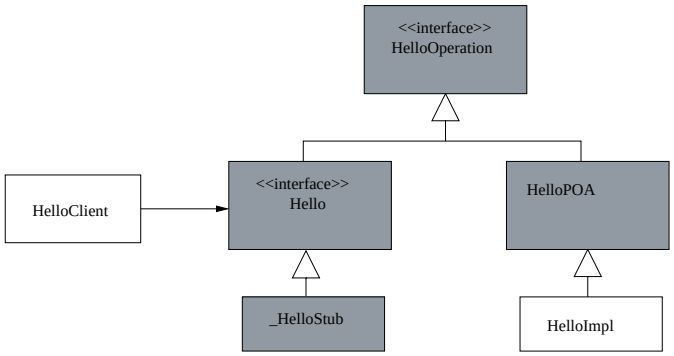

Figure 3.4: The JDK way to structure client and server implementation. Gray classes are generated by the IDL compiler

For an interface called `x`, the compiler produces the files: `xOperation.java`, `x.java`, `xPOA.java` and `_xStub.java`.

xOperation.java : This is a Java interface corresponding to the IDL interface

xPOA.java : This class inherits from the interface `xOperation.java` and contains the skeleton for the actual implementation of the `say_hello` method. As can be seen in the structure overview the actual implementation of the method must be specified in a subclass of `xPOA.java`. Note that Java uses the OMG Portable Object Adapter to implement an object adapter.

x.java : This class also inherits from the interface `xOperation.java`, but here it is to be used on the client side. The client uses delegation to access operations in this class.

_xStub.java : This class inherits from the client side interface, and contains the stub implementations of the operations.

Running this example is done by simply launching the server (this is done with the

command `java HelloImpl`). The server initialises the ORB and activates the object implementation. This causes the ORB to generate an object reference to the object implementation. The client has to get this object reference to be able to call the object implementation[3]. The client also initialises an ORB and with the object reference the client can call the object implementation.

The object reference contains enough information for the client ORB to locate the server ORB and make the call regardless of whether the client and server are placed on different physical machines.

3.4 ORB Interoperability

The ORB is specific to a certain language mapping. We have already encountered the `JDK` ORB with its IDL compiler. This is not the only Java ORB available, for instance Orbacus[4] provides another Java implementation of an ORB.

As we saw in the previous section clients and servers could be running on different ORBs. This is transparent to the clients and servers: the ORB to ORB communication is handled in the ORB core. Since many implementations of ORBs exist there is a need for a standard way for them to interoperate. To handle this need OMG has formulated the ORB Interoperability Architecture.

ORB Interoperability Architecture

The purpose of the ORB Interoperability Architecture is to define a conceptual framework, allowing ORB to ORB communication. In [6, p. 468] this is written as follows: "Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementation". For the architecture to interoperate a protocol is defined, called GIOP.

GIOP and IIOP

In order to make interoperability between ORBs possible it is necessary to specify a standard transfer protocol. This is done by the General Inter-ORB Protocol (GIOP) defined by OMG. It is specifically designed to allow ORB to ORB interaction, and to be implementable over any transport protocol that meets a minimal set of assumptions.

[3] The ORB provides a naming service that eases the distribution of object references (see [5]), but this feature is primarily needed in distributed systems, and is therefore out the scope of this thesis

[4] http://www.orbacus.com

GIOP defines the external data representation (called CDR) used to represent all of the types allowed in CORBA and defines the general transfer syntax of requests and replies.

GIOP defines a general transfer syntax, but OMG also specifies how GIOP is going to be implemented using TCP/IP transport. This implementation is called the Internet-Inter-ORB Protocol (IIOP). The relationship between GIOP and IIOP is the same as that of IDL and a specific language mapping, e.g. the Java mapping. IIOP provides "out of the box" interoperability with other ORBs using the same protocol.

3.5 Example

In section 3.3 there was a description of a Java client to a Java server example, but since language interoperability is the focus here we will investigate the issue of making two different languages interoperate. We are going to change the server side with a C++ server. This can be done by downloading and installing the Orbacus C++ ORB with corresponding IDL compiler. The IDL interface is the same as the one used in the previous example (see page 18)

Using the Orbacus IDL compiler on the interface produces the files: `hello.cpp` and `hello_skel.cpp` with corresponding `.h` files. The `hello.cpp` contains the client stubs and `hello_skel.cpp` the server skeletons, so in our case we are only interested in the latter. To provide an object implementation simply include the file that contains the skeletons and provide the method implementation.

When running the server a file called `hello.ref` is created. It contains the object reference for the Hello object. Copying this reference to the directory where the client is running and launching the client starts the communication[5]. The client invokes the method `say_hello` on the object reference with the argument "World". The result is printed on the screen and it reads "Hello World".

3.6 CORBA Support for Language Interoperability

CORBA does support language interoperability because it enables clients written in one language to call object implementations written in another. But in comparison with the interoperability demands there are many things to mention:

1. **First Class Citizens**: Once a CORBA object is created by the client there is no

[5] Again in a real world example a naming service would be used

telling whether this reference is a local object written in the programming language or if this is a remote object. But how does the client create this object reference? It does so by using the ORB, probably using the naming service to refer to the active instance of the object implementation. It is not as easy to get an object reference to a CORBA object as it is to get a reference to a local object. This is obvious since CORBA objects could be distributed.

Implementational inheritance is not possible. Inheritance is only possible on interface level.

The conclusion must be that there is no first class citizenship of CORBA objects.

2. **One Language, One Compiler**: Clearly this demand is not fulfilled. Interfaces are expressed in IDL and implementations and client usage of interfaces are written in any language supporting CORBA.

   Building either the client or the server side requires the use of two compilers. First the interfaces are compiled using an IDL compiler specialised for the specific implementing language and the ORB provided. With the generated files the programmer can then add the implementation/usage of the interface methods.

3. **Type Safety**: Calling methods on interfaces can happen in two ways. If the interfaces are known at compile time the compiler can perform type checking and the usage will be type safe. If the interfaces are not known at compile time the ORB has the interfaces `Dynamic Invocation` and `Dynamic Skeleton` that adds a reflexion mechanism. Using these dynamic interfaces there is no compile time type checking and it is therefore type unsafe.

The focus of this thesis is language interoperability which means that we have no interest in the distribution parts. And distribution being the most crucial part of CORBA it is perhaps unfair to discard all of this. CORBA does support language interoperability, but with respect to our needs only to a limited extent.

4 Microsoft COM

The Microsoft Component Object Model (COM) [4] is a platform independent and object oriented system for creating binary components that can interact. One important thing to understand is that COM is not an object oriented programming language, but a binary standard. This standard is an object model that specifies how COM objects interact with other COM objects. Even though COM is a specification that could be implemented on any platform it is only available on the Window platforms.

One of the design goals of COM has been "Breaking the monolith". Many applications today have the monolithic form, i.e. one huge binary chunk compiled by the vendor and shipped to the customer. If errors are discovered the vendor must supply a new monolith to the customers. By "Breaking the monolith" Microsoft means that applications should be split into smaller chunks. But when these chunks are compiled into an application this will also be a monolith, so in order to avoid the monolith we must have dynamically linking of the smaller parts.

With dynamical linking it is possible for customers to customise the applications by simply substituting a component for another.

A second goal of COM is to increase reusability of older software. When the components have been made they can be used over and over again by anyone. This not only means reusability but also faster application building.

Now that we have stated some of the benefits of COM lets take a closer look at the object model.

4.1 The Object Model

In COM there is a strict separation of client and component. The only thing they share is interfaces.

The client dynamically loads the component and negotiates with the component what interfaces it can get. The client asks the component to get an object that has a type corresponding to some interface. If the component actually implements this interface,

then it returns an instance of an object with this type and the client can use it.

As stated in the introduction we would like to avoid a recompilation of the client, and since interfaces are a part of the client interfaces must never change – a change would mean a recompilation of the client. As soon as an interface is created it must never change.

Example

A small example will illustrate some benefits of this model. Imagine that we have a code editor in which to write your code in. Indenting is supported in the editor by a COM component that handles this. Suppose now that the provider of the indenter finds that some extra functionality is needed. They provide a new component with some additional functionality, but it is accessed through a new interface. The old interface still exists in the component so the old client uses this component without a recompilation. Figure 4.1 shows this.

Figure 4.1: Clients do not notice that components change as long as they still support the old interfaces

4.2 Details

The above was an overall description of COM, but to understand it in depth some details have to be explained.

As with CORBA there is also `IDL` for COM and it can be used in a similar way (an IDL compiler generates the basic structure and the programmer fills in the details). Having this extra language clearly breaks the second interoperability demand, so if it is possible to avoid it the interoperability evaluation would get a higher rank. For COM there exists language bindings that eliminates the need for `IDL` [19][1]. All programming examples will be given in BETA.

[1] Avoiding IDL could probably also be done with CORBA, but the rationale is that no binding does this (at least not to the knowledge of the author)

Interfaces

The most important interface in COM is *IUnknown*. All other COM interfaces must inherit this interface. Below is an *IUnknown* implementation.

```
IUnknown: COM
  (# QueryInterface:<
       (# id: ^text, IF: ^IUnknown
        enter id[]
        do Inner
        exit IF[]
        #);
      AddRef:< (# do Inner #);
      Release:< (# do Inner #)
  #)
```

As can be seen *IUnknown* contains three methods:

1. **QueryInterface:** QueryInterface is the most important method in *IUnknown*. Loading of a component returns a reference to an *IUnknown* interface. The client does not load the component just to get an *IUnknown* interface, but with this it can query for other interfaces in the component. An id is entered which is a unique identifier of an interface. If this interface is implemented by the component a reference to it is returned. If not `NONE` is returned.
2. **AddRef:** AddRef is a method to be called whenever a component is instantiated. When called the component can then increment the reference counter of the component.
3. **Release:** Release is to be called when a component is no longer used. When called the reference counter is decremented, and if it reaches zero the component can be released from memory.

The two latter methods give a mechanism for managing the lifetime of the component.

Vtable

As said COM is a binary specification. In order to be called a COM interface a special binary layout has to be adhered to. This layout can be seen in figure 4.2. The interface pointer *pIX* points to another pointer, the vtable pointer. This is a pointer to an array

of pointers each pointing to implementations of the methods declared by interface *IX*. Since all COM interfaces inherit from *IUnknown*, pointers to methods `QueryInterface`, `AddRef` and `Release` will always be located in this array. The COM standard demands that these methods always be the first three entrances in the array. With this layout a client can always find the IUnknown methods and thereby query for other interfaces.

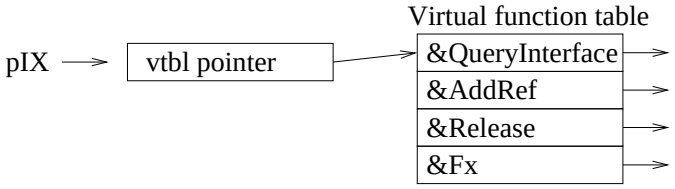

Figure 4.2: The virtual table for the interface *IX*. The first three entries are methods from *IUnknown*. *IX* declares the one method Fx

Details of QueryInterface

Remember that a COM component can implement many interfaces. Each of these interfaces implements *IUnknown*. Confusion in implementing all these interfaces can easily arise. Since `QueryInterface` is responsible for returning all the implemented interfaces care must be taken when implementing the method. The following rules of `QueryInterface` should be followed [4, p. 52]:

1. A component has only one *IUnknown*. If a component implements interfaces *IX* and *IY*, querying *IX* and *IY* for *IUnknown must* result in the same *IUnknown* interface.

2. Given an instance of a component, if this component successfully returned an interface *IX* once it will always succeed. Reversely, if it failed it will always fail. Note that this only applies for instances of components. Interfaces may be added to components subsequently.

3. Reflexivity. Querying an interface for itself will always succeed.

4. Symmetric. It is always possible to get back to the start. Having interface *IX* and successfully query for interface *IY*, will guarantee a subsequent query on *IY* for *IX* to succeed.

5. Transitivity. If *IY* is reachable from *IX* and IZ is reachable from *IY* then IZ is reachable from *IX*.

Note that the last three rules actually implies that `QueryInterface` is an equivalence relation [20, p. 38], (if `QueryInterface` is viewed as a relation[2]). An equivalence relation is a nice algebraic property.

GUID

As mentioned above, for `QueryInterface` to return an interface it has to be given a unique *id* for it. As stated above, interfaces must never change, so the unique *id* will always refer to the same interface.

To identify an interface COM uses a 128 bit structure called the Globally Unique Identifier (GUID). It can be generated by a program available in the Microsoft Visual C++ package, and uses an algorithm that ensures that no two computers would generate the same GUID [4, p. 112].

An example of a GUID: *166769E1-88E8-11cf-A6BB-0080C7B2D682*

An id to reference components uses the same structure as GUIDs. When referencing components the acronym CLSID (class identifier) is used rather than GUID to avoid confusion.

Creation of Components

In the COM library there are methods for loading components. The method to be used when loading components is `CoCreateInstance`. `CoCreateInstance` has the following BETA interface:

```
CoCreateInstance:
  (# clsid: ^text; pIUnknownOuter: ^IUnknown;
     context: @integer; id: ^text; result: ^IUnknown
  enter (clsid[], pIUnknownOuter[], context, id[])
  <<SLOT CoCreate: doPart>>
  exit result[]
  #)
```

clsid : unique id for the component to be loaded.

pIUnknownOuter : argument used when the component is being aggregated. This will be described in the next section.

[2] *IX* and *IY* are said to be related if *IX*.`QueryInterface`(*IY*) succeeds

context : an integer specifying whether the component is executed in-process, local or remote. The use of this argument is not elaborated further since it is beyond the scope of this paper.

id : a GUID identifier of the interface wanted by the client. This quickens the process of getting the wanted interface. The default interface to return is *IUnknown*, but then the client would have to query the *IUnknown* for the interface wanted.

CoCreateInstance uses the CLSID to find the identified component. This is done by using the Windows Registry. The Windows Registry stores the names of the DLL containing the component with the CLSID.

4.3 Containment and Aggregation

COM does not support implementation inheritance. Having inheritance i.e. component B inherits from component A would mean that component B is dependent on the *implementation* of component A. This is, of course, a strict violation of the basic rules of COM because the implementation of a component could change. Inheritance is not possible in COM.

In order to simulate inheritance in COM two programming paradigms exist, namely containment and aggregation.

Containment

Containment is when one component *contains* another. The component that contains the other is called the outer component and the contained component is called the inner.

The outer component declares all the methods of the contained interfaces. Clients only talk to the outer component and do not know or care whether it has a contained component or not. The outer component can choose to simply delegate the method invocation to the inner component without adding any code itself, or to add code around the invocation of the inner component.

Containment poses no problems in the COM object model. The outer component has pointers to the inner components interface. These pointers are created on the construction of the outer component. Should creation of the inner component fail the whole outer component must also fail. If it succeeds the outer component can invoke methods on the inner.

The inner component does not have to know it is begin contained by another. The outer component acts as a normal client to it.

Aggregation

In aggregation the interfaces of the inner component are handled directly to the client. This means that once the client gets the interface there is no connection to the outer component anymore.

Aggregation saves the outer component the trouble of all the forwarding and declaring of methods and interfaces supported by the inner. But this comes at the expense of other complexities.

This first idea of how to implement aggregation is illustrated with the following example. Suppose the outer component implements interface *IX* and the inner component implements interface *IY*. The `QueryInterface` of the outer component returns interface *IX* when queried about it. But if queried about *IY* the outer component forwards the query to the inner components QueryInterface and returns the result to the client. This solution may seem easy and sound, but in fact it conflicts with the demands stated about `QueryInterface` above.

Problem With Aggregation

Since all interfaces inherits from *IUnknown* the method `QueryInterface` is implemented in both the inner components interfaces and outer components interfaces. The following example illustrates a violation of the symmetric demand on the `QueryInterface`.

The problem was already apparent in the example given above: the outer component implemented *IX* and aggregated *IY*. This means that querying *IX* for *IY* succeeds. But with *IY* the query for *IX* will fail. The inner component has no knowledge that it is being aggregated and therefore must return an error when queried about *IX*. The solution violates both the `QueryInterface` rule about symmetry, but also the one about *one* implementation of *IUnknown*.

This is a situation that should not happen. Clients should not be aware of aggregated components, but should believe that they are communicating with only one component. This means that simply returning the interface pointer from an inner component is not good enough.

Solving the Problem

In the problem of the two different *IUnknown*s, the inner component must somehow be able to return the *IUnknown* for the outer component, when it is being aggregated. This would solve the problem.

The solution to the problem is to have a delegating *IUnknown* along with the old *IUnknown* implementation in the inner component. The delegating *IUnknown* is called whether the component is being aggregated or not, but delegates to its own *IUnknown* implementation when not being aggregated and delegates to the outer components *IUnknown* implementation when being aggregated. This is shown in figure 4.3.

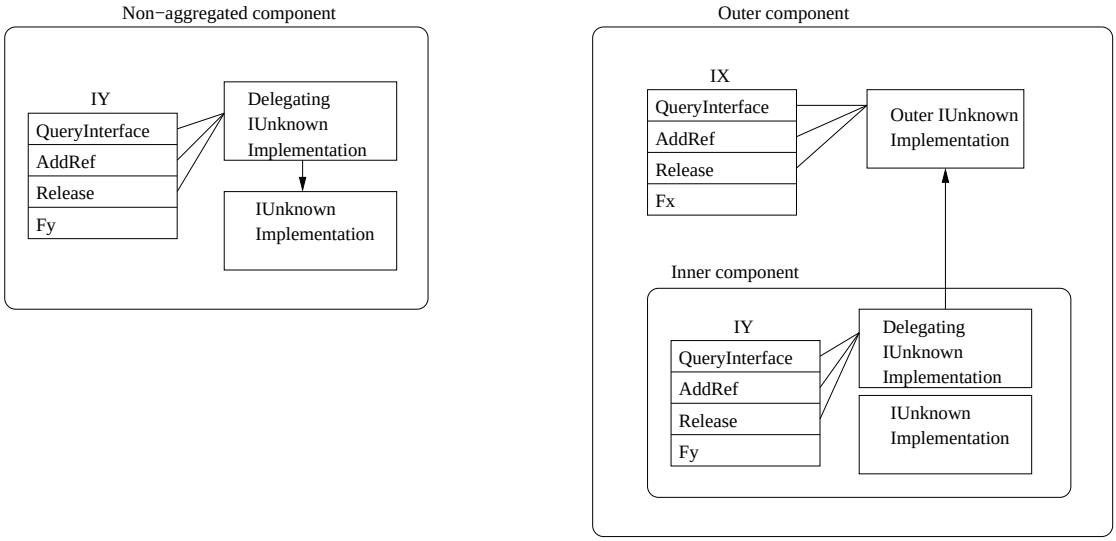

Figure 4.3: On the left is the non aggregated component. The delegate uses the regular implementation of *IUnknown*. On the right is a component aggregating interface *IY*. The aggregated component uses the outer *IUnknown* implementation

Setting up the delegate *IUnknown* is done when the component is created. The creation happens by calling the method `CoCreateInstance`. Remember the parameter `pIUnknownOuter`. If this parameter is *null* then the component is not being aggregated and the implementation of *IUnknown* should therefore be its own. If it is non *null* then it is being aggregated and the delegate *IUnknown* uses the given *IUnknown* pointer when delegating.

Having two implementations of *IUnknown* increases the complexity of writing a component. The important thing to note is that the inner component has to be aware of the fact that it can be aggregated. It is possible to write a component that exposes a few

interfaces of a large component – but the large component has to be changed accordingly. The bottom line is that it can be done, and also it generalises (the component containing IX and aggregating IY could be aggregated by another component), but with a large amount of extra code.

Implementing Inheritance using Containment and Aggregation

Inheritance is not supported by COM, but containment and aggregation are programming techniques that allows for an equally strong programming model called delegation [21]. We will not go into detail with this, but just state that by adding statements around the invocation of the method in the inner component the relationship between the outer and inner component seems like an inheritance relation. The difference between containment and aggregation is that in containment the contained component only knows itself, whereas an aggregated component knows its aggregator. This difference is illustrated in figure 4.4.

Figure 4.4: The difference of containment and aggregation when it comes to self reference. Containment is shown on the left – the inner component knows nothing of the outer and can only call itself. On the right the aggregated (inner) component knows of the outer and calls to itself will go through the outer

4.4 Automation

This section will briefly cover a different way for clients to communicate with components called Automation. Automation is a very big subject so many details are omitted here. Automation gives interpreted and macro languages an easier access to components, but also makes it easier for these languages to write components [4, p. 279].

Automation is built on top of COM, and facilitates a looser binding of component and client. The looser binding is achieved with an interface allowing method invocation to happen by sending a textual representation of the name. This eliminates the compile-time type checking, and has an additional penalty of invocation time for the various methods.

Automation introduces some new concepts listed below:

IDispatch: The interface supporting the method invocation by name is called *IDispatch*.

Automation Server: A COM component implementing the interface *IDispatch* is called an Automation Server.

Automation Controller: An Automation Controller is a COM client that communicates with an Automation Server through its *IDispatch* interface.

The main new thing here is the special interface *IDispatch*. The two most important methods in IDispatch are `GetIDsOfNames` and `Invoke`. Below these are shown in BETA (the IDL type of the arguments is shown in comments).

```
GetIDsOfNames:<
  (# result: @int32;
     riid: ^GUIDdata;    (* IID& *)
     rgszNames: @int32;  (* LPOLESTR* *)
     cNames: @int32u;    (* UINT *)
     _lcid: @int32u;     (* LCID *)
     rgDispId: @int32;   (* DISPID* *)
  enter (riid[], rgszNames, cNames, _lcid, rgDispId)
  do INNER;
  exit result
  #);
Invoke:<
  (# result: @int32;
     dispIdMember: @int32;               (* DISPID *)
     riid: ^GUIDdata;                    (* IID& *)
     _lcid: @int32u;                     (* LCID *)
     wFlags: @int16u;                    (* WORD *)
     pDispParams: ^struct_tagDISPPARAMS; (* DISPARAMS* *)
     pVarResult: ^VARIANT;               (* VARIANT* *)
     pExcepInfo: ^struct_tagEXCEPINFO;   (* EXCEPINFO* *)
     puArgErr: ^int32uHolder;            (* UINT* *)
  enter (dispIdMember, riid[], _lcid, wFlags, pDispParams[],
```

```
        pVarResult[], pExcepInfo[], puArgErr[])
do INNER;
exit result
#);
```

As can be seen from the methods they take many arguments, and it is complicated to use them. `GetIdOfNames` is a mapping from a human readable name to an integer. This integer is one of the arguments given to the `Invoke`-method allowing it to locate the implementation. The arguments to the method to be invoked are also given to `Invoke`. Knowing only the name of a method of an *IDispatch* method, it is possible to call this method.

In the regular COM model clients accesses component methods through the interface's vtable pointer. With IDispatch clients communicate with the IDispatch interface to get the local method id, which is just an integer. By calling invoke with the correct set of arguments for the method to be invoked the invocation will succeed. Type checking is not possible in this form of component access.

Automation covers a large area and allows for features that are not in the scope of this thesis.

4.5 COM Support for Language Interoperability

COM components could be targeted by any compiler, which means that methods accessed could have originated from any language – hence COM supports language interoperability.

In comparison with the interoperability demands the following can be noted:

1. **First Class Citizens**: Inheritance is not supported. So a COM interface is not a "first class citizen" when used from a different language. Instantiation of COM objects is also different from instantiating local objects. COM objects are instantiated by calling the method `CoCreateInstance`. But other than that COM objects can be used as if they were local objects.

2. **One Language, One Compiler**: As for the expression in one language COM can support that. IDL is present in COM and for some language bindings it can probably not be ignored – but there are also language bindings where it is possible to use only one language and one compiler.

3. **Type Safety**: Basically COM is type safe. The interfaces referenced in the code is type checked against their declarations that have to be included. In C++ interfaces are described as header files and in BETA they are expressed with virtual pattern declarations.

   The use of the QueryInterface must be done with care. The method returns an IUnknown pointer, which has to be cast correctly by the user.

   The IDispatch interface does not make things better. With this interface the user just sends a string with the name of the method and arguments to invoke to the component. There is no compile time knowing of whether the method exists or if the arguments are correct.

Overall COM offers means to express languages interoperability in an almost ideal way. The only demand not satisfied is implementational inheritance. The lack of type checking in relation to IDispatch is not regarded as breaking the demand, since IDispatch and automation is an extension to COM and users can ignore this.

5 Execution Environments

Execution environments, or platforms, provide a standardised environment for code execution. Two such examples of execution environments are the Java Virtual Machine (JVM) and Microsofts Common Language Runtime (CLR).

The Java language system was developed and released by Sun Microsystems in 1990 [10]. This language system defines the object oriented programming language Java implemented by the Java compiler. The Java compiler converts Java source files into Java `class` files – binary platform-independent files containing Java bytecodes. Java bytecodes are Java instructions to be interpreted by the JVM.

The Microsoft .NET framework was designed and released by Microsoft research in 2000 [10]. This framework contains many components, but the primary is the run time environment called the Common Language Runtime. Like the JVM this execution engine is capable of interpreting an intermediate form. This intermediate form consists of bytecode instructions called Common Intermediate Language codes – the .NET pendant to Java bytecodes. A CLR compiler converts language source files into `assembly` files, which can then be interpreted by the CLR.

5.1 Java Virtual Machine

The JVM is designed and developed in connection to the Java language [3], and it is solely designed as a platform to be targeted from Java sources. The object model of Java is therefore reflected in the JVM, which offers direct support for some of its language features, e.g. classes are allowed to inherit from one super class and implement several interface classes and it is only possible to have a single return value of a method.

Some of the important properties of JVM are:

- It is based on an evaluation stack, which can be manipulated by the Java bytecodes. Method arguments are pushed to the stack before invocation of a method, and when completed the return value is located on the stack.

- It is strongly typed. Instructions are specific to certain types (see example below). Pointer manipulation is not allowed.
- It is garbage collected. Objects no longer referenced are automatically released.

The Instruction Set

In the JVM instruction set, one instruction consists of one byte, called a Java bytecode. There are 256 different Java bytecodes. The bytecodes are stored in big-endian order.

As an example consider the set of bytecodes below that loads two local variables and adds them, which is done using the stack. Next to each bytecode the contents of the stack after the bytecode is interpreted is shown. The `...` symbolises the content of the stack immediately before the bytecode `iload_1` is reached.

```
iload_1     ; ..., local1
iload_2     ; ..., local1, local2
iadd        ; ..., local1+local2
```

Note the *i* in front of each instruction. This indicates that the types of the values manipulated on the stack are integer. A similar program adding two `double`s would use instructions prefixed by *d* instead of *i*.

5.2 Common Language Runtime

Unlike JVM, the CLR is designed as a general language platform, i.e. designed to be targeted by any programming language. This is mirrored by the instruction set that supports what Microsoft calls the Virtual Object System. This object model supports three kinds of methods (static, instance or virtual instance), and various method invocation styles. The support for different invocation styles is necessary for efficiently implementing language support for e.g. ML – CLR instructions has support for tail recursion calls.

The properties of the CLR are:

- It is stack based as the JVM.
- It is strongly typed.

- It is garbage collected, but it is possible to define code to be `unmanaged`, i.e. not garbage collected.
- It supports many different method invocation styles.

The Instruction Set

Like JVM, the instruction set of CLR consists of bytecodes – i.e. each instruction has the size of one byte. The instruction set of CLR contains 220 bytecodes, and the byte ordering is little endian.

Below is a small example of bytecodes that loads two variables and adds them. Again the stack is shown on the right of the bytecodes.

```
ldloc.1      ; ..., local1
ldloc.2      ; ..., local1, local2
add          ; ..., local1+local2
```

Notice that the instructions are polymorphic – there is no type information encoded in the bytecodes. The same set of bytecodes could just as well add two `doubles`. In Java bytecodes the type of the variable manipulated is specified in the bytecode. For the type information to be encoded in the bytecode the compiler has to resolve the types at compile time in order to be able to emit the correct bytecodes. This means more work at compile time, but a faster interpretation. The polymorphic bytecodes mean lesser work on compile time, but postpones the burden until run time [9].

5.3 Comparison

There are both many similarities between JVM and CLR and many differences. They are both stack based, strongly typed and garbage collected virtual machines. But JVM is designed as a platform for specifically supporting Java and CLR is a general language platform. This difference is seen in CLRs variety of method invocation styles, as mentioned, and the possibility to allow for unmanaged code. Some language idiosyncrasies could provide difficulties or performance penalties when porting to JVM. An implementation of C on JVM and an implementation on CLR is discussed in [11], it is concluded that implementing C on JVM is very unpractical, and it was only done to some extent. The sizes of the programs generated was in average twice the size for the JVM platform than for CLR. This shows that when it comes to languages other than object oriented CLR is superior to JVM.

Instead of interpreting the CIL bytecodes the CLR performs a compilation to the native platform. CLR bytecodes are thus never interpreted. This difference is a technical detail of the .NET platform, and is not important when porting a language to CLR. JVM is also able to compile to native code, but this is a new feature of JVM and has been added to increase performance.

In [1] the porting of BETA to both JVM and CLR is described. It is concluded that the port was equally difficult. As mentioned CLR is designed to be targeted by any language, and should have several advantages over JVM in this respect [9]. But, apparently, when it comes to object oriented languages the difference is smaller.

5.4 JVM & CLR Support for Language Interoperability

The interoperability possibilities with the two execution environments CLR and JVM are for compilers of different languages to target these platforms. For example: assume the two different languages A and B are ported to JVM. It is then possible to compile code written in the language B to JVM, lets assume that the compiled B code declared a class `BClass`. From the language A it is now possible to write a class `AClass` that inherits from `BClass` – this inheritance is expressed as inheritance would be expressed in language A even though it is interoperability between language A and B. With this form of interoperability the programmer is not able to distinguish interoperability with foreign languages from usual programming in this language.

To be able to use CLR and JVM as interoperability platforms requires porting of languages. This port is not a trivial task and the CLR is superior when it comes to functional or unmanaged languages [11], but when it comes to object oriented languages CLR and JVM are equally hard [1].

In respect to the ideal demands on language interoperability JVM and CLR turn out to be optimal:

1. **First Class Citizens**: Objects and classes from other languages are indeed first class citizens. Classes compiled from other languages cannot be distinguished from classes written in the target language – the language barriers have disappeared.

2. **One Language, One Compiler**: CLR and JVM support many languages, and given one of those it is possible to use only this because the compiled result of the other languages are language independent. This means that everything is expressed in one language and compiled with one compiler. See [1] for implementations of compilers on JVM and CLR.

3. **Type Safety**: Since JVM and CLR are strongly typed interoperability with other languages will have to follow the type rules of these platforms and will therefore be type safe.

Since CLR and JVM supports the author's definition of optimal interoperability it is appropriate to investigate the use of them as such in practice. A `Hello World` example cannot really illustrate the benefits and disadvantages of these standards, so there is a need for a more "real world" example. Implementing applications using a framework is such an example. A framework defines the skeleton of the applications to be created by use of it. A successful usage of a framework implemented in a language different from what the framework expects would certainly show the interoperability to have a great practical aspect.

The case study of how to implement language support in a general IDE platform will constitute the second part of this thesis.

5.5 IDE Platforms

The two widespread IDE platforms currently available are Eclipse and The Microsoft Visual Studio .NET[14] (VS). They are both designed to be extensible IDE platforms with ability to support any languages IDEs.

The ways to write plug-ins for both Eclipse and VS are somewhat similar. In Eclipse plug-ins consist of Java class files assembled in JAR libraries. In VS plug-ins are created by implementing COM interfaces, but with the newly announced *managed VSIP API*[15] it is possible for plug-ins to be CLR bytecode assemblies – making the plug-in architecture conceptually the same as that of Eclipse.

The benefits of using IDE platforms are many. The IDEs are all located in the same program, which means that the menus, toolbar and all layout have the same look for all IDEs – when a new plug-in IDE is installed the operations for opening files and browsing the source files are the same making a new IDE easier to use. The maintenance burden of the implemented IDEs is smaller since the framework is maintained by Eclipse or Microsoft. Finally the distribution is made easier since only a plug-in has to be installed, not the whole program.

What makes these IDE platforms interesting from the point of view of language interoperability is when it comes to implementing IDEs. Most languages already have an IDE, but usually as a stand alone application. The benefits described above shows it is an advantage to have the IDE in an IDE platform, both for implementors and users of

the IDE. It is, however, not desirable to reimplement the whole IDE making it suit the specific framework. The question is then: how can language interoperability ease the creation of a language IDE in platforms such as Eclipse and VS when a stand alone IDE is already implemented? The rest of this thesis focuses on the interoperability issues encountered when implementing support for a BETA IDE in Eclipse.

6 Eclipse

Eclipse.org is a consortium initially formed in November 2001 by companies Borland, IBM, MERMANT, and others[1]. Later many other companies joined and today Eclipse.org has more than 30 different companies as members.

The purpose of Eclipse.org is to develop a universal tool platform and IDE i.e. a programming platform that unifies product building (code writing, debugging, versioning etc.) in one application called Eclipse (sometimes Eclipse platform). Some of the design goals of Eclipse are that it should be available on many operating systems, support tools for manipulating many different content types, be extendable and be written in Java.

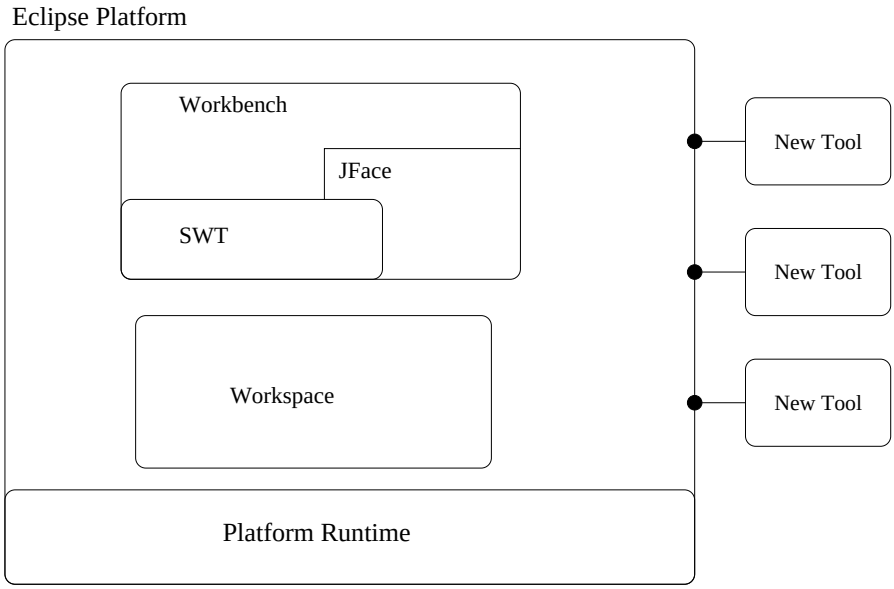

Figure 6.1: Conceptual overview of the Eclipse platform with some of its APIs

Figure 6.1 shows a conceptual view of Eclipse. Eclipse is an extensible platform architecture and extensions are called plug-ins (labelled "new tool" in the figure). Figure 6.1 shows some of the APIs available in the Eclipse platform, i.e. SWT, JFace, Workbench

[1]See http://www.eclipse.org/org/index.html for the complete list

and Workspace, these APIs are implemented as plug-ins. Before going into detail with these APIs it is important to understand the plug-in nature of Eclipse.

6.1 The Plug-in Model

When contributing plug-ins to Eclipse it is always done by extending existing Eclipse plug-ins – this process is bootstrapped by some core plug-ins. Each Eclipse plug-in is deployed in its own directory in the Eclipse installation and it logically consists of three parts:

The manifest an XML file describing how this plug-in interconnects to other plug-ins (i.e. what plug-ins it extends, what plug-ins it is dependent on, and plus extension points defined) and where relevant code for this plug-in is located.

The code the code that implements the actual functionality of the plug-in. Code connected to Eclipse plug-ins consists of Jar libraries of Java class files.

Resource files other files needed by the plug-in, e.g. images or icons.

In a manifest it is possible to declare two types of extensions: extension-points or extensions. Extension-points are called "host plug-ins", and defines a new hole to be filled by extension plug-ins. Extensions are called "extender plug-ins" and fills a hole of a host plug-in. With these new concepts it is possible to give a more precise view of Eclipse. Figure 6.2 shows how the APIs are plugged into the platform runtime of Eclipse.

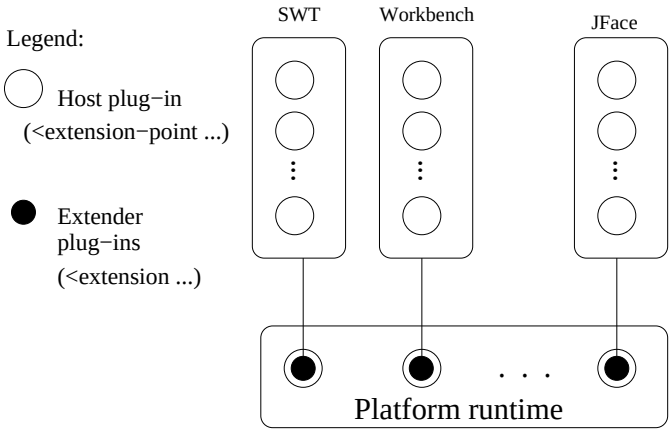

Figure 6.2: Eclipse architecture. The APIs are plugged into the Eclipse platform runtime

Tool builders contribute to Eclipse by wrapping their tools in plug-ins as shown in figure 6.3.

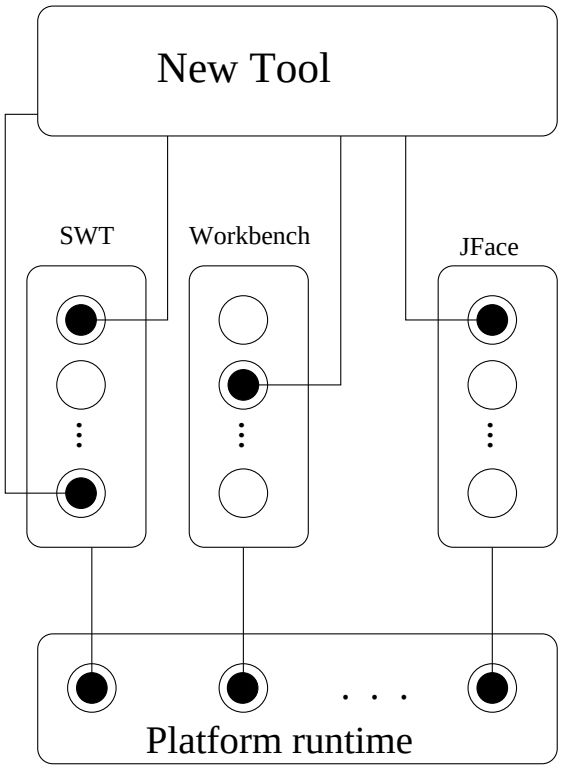

Figure 6.3: New plug-ins extends existing (host) plug-ins

Host Plug-ins

A host plug-in is declared in its manifest file. The following example is from a host plug-in provided by the Eclipse UI API called `org.eclipse.ui.actionSets`. This plug-in gives extenders a possibility to add menus, menu items or buttons to the toolbar of the workbench of Eclipse.

The XML declaration looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
 id="org.eclipse.ui"
    name="Eclipse UI"
    version="2.1.0"
    provider-name="Eclipse.org"
    class="org.eclipse.ui.internal.UIPlugin">

 <extension-point id="actionSets" name="Action Sets"
```

```
      schema="schema/actionSets.exsd"/>
    <!-- Other specifications omitted. -->
</plugin>
```

This declaration has to have a unique id, for extenders to reference it. By convention Eclipse ids are constructed like Java package names, here `org.eclipse.ui`. The important part of this declaration is the `schema` attribute. This points to a `exsd` file, i.e. an XML file declaring how extender plug-ins of this host plug-in should be structured.

The `class` attribute is the fully qualified reference to a class with the knowledge of how to process the extensions to this plug-in, in this case it is called `UIPlugin`. On Eclipse start-up all the manifests are parsed and placed in a plug-in registry that can be accessed programmatically through a plug-in registry API. By using this API the class `UIPlugin` can query for the extensions of itself, and by reading their values configure the buttons or menus declared by them.

Extender Plug-ins

Extender plug-ins fill the plug-in defined by a host plug-in. In chapter 7 the implemented BETA IDE plug-in for Eclipse is described, and its manifest consists of many extender plug-ins. Extending a host plug-in is done with the plug-in manifest file. The example below is taken from the BETA IDE manifest, and it implements an action (a JFace metaphor see section 6.2) which can be activated either by a toolbar button or from a menu.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  <!-- Other specifications omitted. -->

  <extension point="org.eclipse.ui.editorActions">
    <editorcontribution
       id="daimi.betaeditor.actioncontribution"
       targetID="daimi.BetaEditor"
       <action
          id="daimi.betaeditor.compileaction"
          label="Compile"
          icon="icons/compile.png"
          tooltip="Compile the edited file"
          class="daimi.util.CompileAction"
          toolbarPath="Normal/additions"
          menubarPath="Project/additions">
```

```
      </action>
    </editorcontribution>
  </extension>

</plugin>
```

To add menus, menu items or buttons to editors you have to extend the point `org.eclipse.ui.editorActions`. In this extension point you specify what the editor contribution should be in the `editorContribution` tag. This tag specifies to what editor the contribution should apply to in the `targetID` tag. The `action` tag is more complicated because it completely describes where the toolbar button is placed in the toolbar and what menu the menu item should appear in. The attributes are explained below:

id unique id denoting this action

menubarPath specifies the menu where this menu item should appear

label the label of the menu item

toolbarPath specifies where in the toolbar this button should appear

tooltip when positioning the mouse over this toolbar button this text is shown.

icon the icon on the toolbar button

class the uniquely qualified reference to the class implementing interface `IEditorActionDelegate`. This class is in plug-in terminology called the "callback object" for this extension.

Callback Objects

A callback object is the connection between the host plug-in and the extender plug-in.

In the example of the host plug-in and the extender plug-in given above the callback object was an instance of the class `daimi.ui.NewAction`. This object is managed by the host plug-in – implemented in the class `UIPlugin`. So this class asks the plug-in API for all extensions of itself, and if we have installed the extender plug-in it will get this declaration. In the declaration it can read the class name of the callback object and thereby instantiate it. When the user then presses the toolbar button the action code of the callback object is executed.

For optimisation reasons there is no need to instantiate the callback object before it is used. Therefore a lazy loading of the objects happen. A small proxy object is created instead of the real. The first time a call is made to the proxy it instantiates the real object and delegates the call[17].

6.2 Eclipse Concepts and API

The concept of Eclipse was shown on figure 6.1. Here it was illustrated how plug-ins (noted as "new tool" on the figure) extended the platform. The white boxes inside of Eclipse (Workbench, Workspace, ...) denote some of the most fundamental APIs of the platform and will therefore be explained in detail.

Workspace

The workspace in Eclipse is the location of the files that the tools are working on. If Eclipse is installed on Windows in `c:\eclipse` then the workspace is placed in `c:\eclipse\workspace`. Every file in the workspace must be associated with a project where each project has its own sub folder in the workspace directory.

SWT, JFace & Workbench

The Eclipse workbench is what provides the overall structure of the Eclipse Platform User Interface (UI) (see figure 6.4) and presents the extensible UI to the user. The workbench is implemented using two graphical toolkits, namely SWT (Standard Widget Toolkit) and JFace.

Since the workbench provides the UI structure the main window the user sees when launching Eclipse is also called the workbench. The figure below shows the workbench with a Java file open.

SWT

SWT is a common operating system independent API for graphical widgets. Even though SWT is platform independent it is tightly integrated with the underlying native window system. This is accomplished by defining a common API to all the supported window systems (e.g. Motif, Windows and MacOS). In a specific native window system

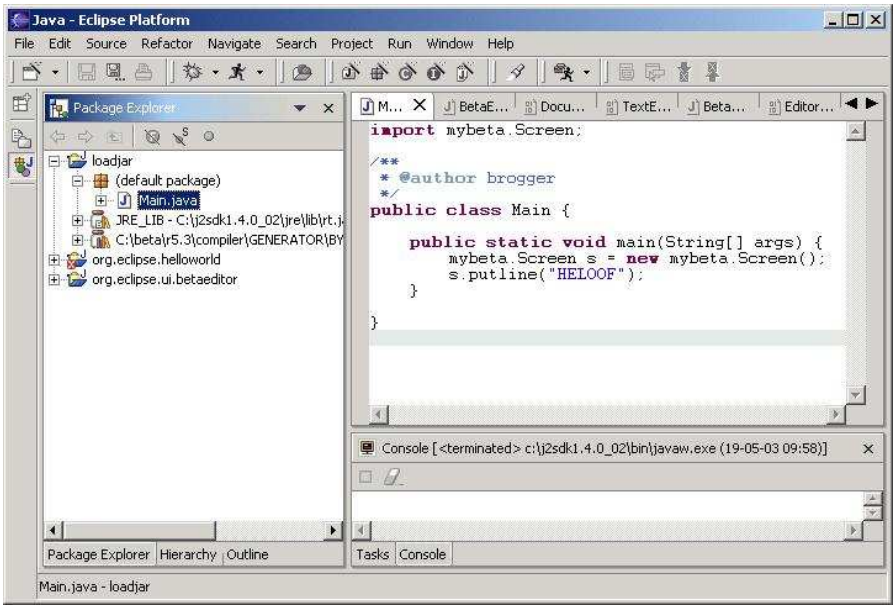

Figure 6.4: Eclipse workbench is what you see when starting Eclipse

SWT uses the native widgets whenever possible. If no native widget is present SWT provides an emulation.

This is unlike Java's AWT which defines a common API for all platforms, and only allows integration with native widgets on the underlying window system. AWT is programming to the least common denominator of the set of supported window systems and its functionality is limited to very simple widgets such as lists, textfields and buttons.

AWT consists of the same Java code on all platforms, and has different layers of C code on each platform. In SWT it is structured differently. On every platform there is a distinct Java implementation which fits the underlying window system. There is a one to one mapping between the Java native methods and the operating system calls, i.e. a Java native method only performs one call to the operating system. Having such a structure makes bug fixing easier than in the AWT case where you have a large amount of C code – crashes in C code should be more identifiable in SWT than in AWT.

JFace

JFace is a higher level UI programming toolkit and is inherently OS independent. It is designed to work with SWT and has UI components such as image and font registries, dialog, preferences etc. Two of the more profound abstractions are actions and viewers.

Actions : An action is used to define user commands without knowing its whereabouts in the UI. An action comprises commands triggered by the user via a button, menu item or item in a toolbar. Since the action has not got any information on the exact whereabouts in the UI it is easier to use the action in many different places in the code. I.e. you could have the exact same action code working as both a button one place and as a menu item another.

Viewers : The JFace viewers work as adapters for SWT widgets. Viewers also add features to the widgets making it possible for clients to add elements of the client domain to the viewer. The viewer is then responsible for keeping the client model and the SWT widget in sync.

Workbench

The workbench is another word for the Eclipse Platform UI. It defines the structure of how the tools interact with the user.

Three concepts exist in the Eclipse Platform UI, namely editors, views and perspectives. Editors are used to open, edit and save files and can contribute to the workbench, i.e. add a new menu item in a certain menu when the current editor is active. Plug-in writers can create their own editor – an extensive example is given in chapter 7.

Views display information about an object. A concrete example is the outline view connected to the Java editor. It displays a list of all methods, attributes and instance variables in the Java file currently being edited. Clicking on an element in the list highlights the corresponding code.

A perspective is simply a certain configuration of editors and views. The user can easily switch between perspectives and thereby choose the perspective that suits his task the best. The example here is Java programming versus Java debugging. When debugging you want a view that shows the call stack, but that view is superfluous when programming, so two perspectives are needed for the two tasks.

6.3 Plug-in Example

As Eclipse is a IDE platform, this example will explore the initial steps in creating an IDE, by creating a text editor that gets activated when files with a certain extension is opened.

The plug-in is called `exampleEditor`, which means that a folder with this name is

located in the `plugin` folder of Eclipse.

The Manifest

Every plug-in needs a manifest file called `plugin.xml`. For this example the manifest contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="exampleEditor"
   name="example"
   version="1.0.0">

  <requires>
     <import plugin="org.eclipse.ui.editors"/>
  </requires>

  <extension point="org.eclipse.ui.editors">
     <editor
        id="example.editor"
        name="A Simple Editor"
        extensions="exa"
        icon="example.gif"
        class="org.eclipse.ui.editors.text.TextEditor">
      </editor>
  </extension>

</plugin>
```

The manifest contains three tags:

plugin : Every plug-in must be specified in the body of a `plugin` tag. The tag has the attributes: id, name and version. These attributes specify a unique identifier for the plug-in, an associated name, and a version number.

requires : This tag declares what external `jar` file to include in the Java `classpath` when running the plug-in.

extension : An extension tag must be specified whenever an extension point is extended. For this particular extension point the programmer has to specify the inner tag `editor`. This tag has the following attributes:

id : A unique identifier for this extension point.

name : A name related to this new editor. The name is then added to the list of different editors.

extension : The file extension that this editor should respond to, i.e. when opening a file with extension `.exa` this editor is used.

icon : The icon that appears on files with extension `.exa`, when using the Eclipse file browser.

class : This attribute points to the class that implements this editor. In this example it points to `org.eclipse.ui.editors.text.TextEditor` which is a class file in one of the Eclipse APIs that implements a simple editor. This explains the `requires` tag earlier – in order to make the JVM see the class `TextEditor` on run time, the `requires` tag states where to find the jar library containing the class file.

So you create a plug-in manifest file defining an extender plug-in that fills extension point `org.eclipse.ui.editors`.

Running the plug-in

The folder of this plug-in is called `exampleEditor` and, as mentioned, it contains the manifest file `plugin.xml`. But the `icon` attribute referred to a file named `example.gif` so this file must also be present in the folder. With these two files in the folder the plug-in can be activated. On figure 6.5 a screen-shot of the running plug-in is shown.

General Thoughts

This example plug-in is somewhat different from how other plug-ins look because there is no code. There is, of course, the code that implements the editor, but this is some of the internal Eclipse code. Usually plug-ins include some additional code.

In the plug-in there could have been an implementation of a text editor. This is done in the BETA editor for the BETA IDE described in chapter 7. The implementation of the BETA editor extends the class `TextEditor` and makes it specific for editing BETA files.

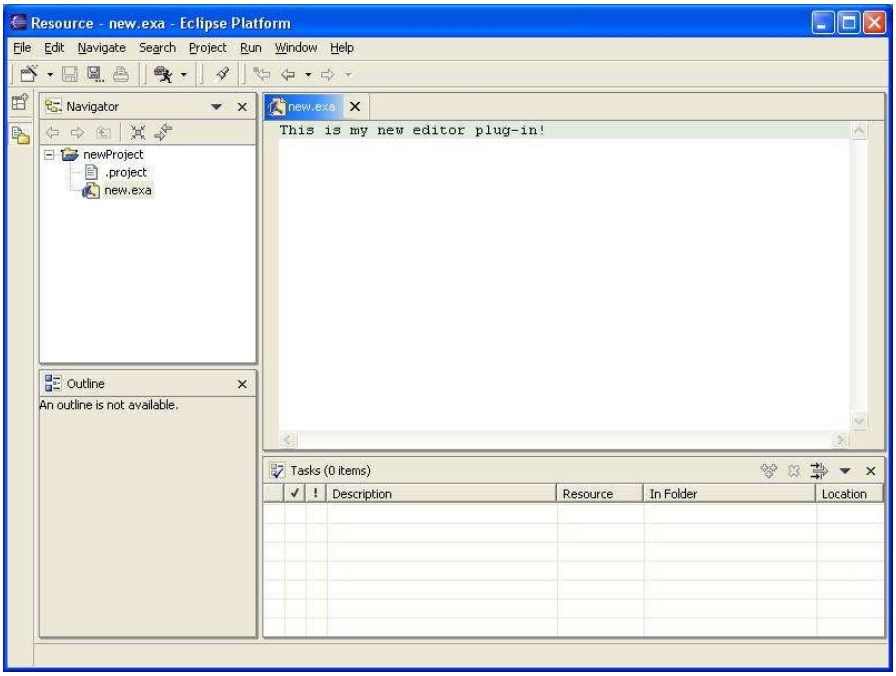

Figure 6.5: The example editor running. Notice the special icon appearing next to the file in the file browser

6.4 Evaluation

Having plug-ins consist of both an XML manifest file and Java code makes the Java code simpler. The dependencies are described in the manifest which is simpler because XML is a simple format. As an example think of writing an editor. Typically one may want to add some buttons to the toolbar. This can be done through the manifest, and all one has to provide in Java code is the actual functionality.

The downside is that the XML is source level - one does not compile the manifest file and therefore one has to run the plug-in before knowing if there is a typo or semantic errors in the manifest. Eclipse provides a graphical plug-in editor, that helps write the manifest file (called Plug-in Development Environment (PDE)), but errors are still not entirely eliminated.

An advantage of the Eclipse plug-in model is that if you have many large plug-ins that you do not use, they do not use any of the machine's memory. On the other hand instantiating a large plug-in can be time consuming, so starting up a new tool can freeze Eclipse for several seconds.

Another advantage of this plug-in model is that reading the manifest files and building the registry is not that time consuming compared to loading the actual Java code. This makes for a faster startup and avoids plug-in activation order problems.

6.5 Eclipse Interoperability

In a framework as Eclipse, language interoperability is a key issue. Eclipse has as design goal a wish to be extendible with any language tool possible. The question is: do you have to write the plug-in in Java? Yes and no. Eclipse requires that the plug-in is written in Java, but by use of JNI it is possible to interface other languages such as C, thereby allowing integration with tools written in C.

Basically, interoperability in Eclipse boils down to interoperability in Java. More precisely interoperability in JVM. The ways to attack integration of other languages with Eclipse is either through JNI or by making your compiler generate Java bytecode.

One could question the choice of JVM as the execution environment of Eclipse plug-ins. The choice of an execution platform is clever Because, as argued in chapter 5, they support ideal language interoperability and can be viewed as language platforms. What about Eclipse using CLR as the plug-in platform? CLR is better suited to be targeted by functional and imperative languages than JVM. Using CLR as run time environment would restrict Eclipse to be run on Windows platforms currently – whereas Eclipse is available on any platform supporting Java, i.e. virtually any. And recall that one of the design goals of Eclipse is that it should be available on many platforms.

The title of this section being “Eclipse Interoperability”, how about evaluating Eclipse with respect to the interoperability demands? Eclipse could be thought of as interoperating with the added plug-ins. Plug-ins must have an XML manifest file, which means that the demand of having one language is not meet. Type safety cannot be met either. Plug-in developers must wait until run time to find out whether the type of the class specified in the manifest follows the contract defined by the extension. We have argued that the advantages of having the XML in combination with code for plug-ins, but it is worrying to conclude that Eclipse supports language interoperability very poorly.

7 MjolnerTool Plug-in

Having examined the Eclipse platform with minimal plug-in examples in the previous chapter it is now time for a "real" plug-in. This chapter describes the BETA IDE Eclipse plug-in.

7.1 The MjolnerTool

For more than twenty five years BETA related development has been done – both at the computer science department of Aarhus University and the company Mjølner Informatics A/S[1]. In the early years it was mostly concerned with writing compilers for the many different supported platforms. As the interest for the language grew, both scientific and commercially, the need for more than just a compiler arised. Mjølner Informatics A/S lead the development and implementation of libraries and tools to help the end users of BETA. The tools grew in number, and to mention a few there were a debugger, semantic editor, UML editor and user interface builder. Instead of having to run these different tools whenever they were needed the idea of having one application that contained all of these tools was conceived – it should be an application that allowed writing, debugging and running code or said with an acronym; an IDE (Integrated Development Environment). The BETA IDE was dubbed MjolnerTool.

Having an IDE is a good thing – the user uses only one application that does it all. This is clearly easier than having several. But why not generalise this idea of collecting applications in one? Instead of having many IDEs, each for a specific language and as a stand alone application, combine the different language IDEs in one application – often referred to as "platform". An example of such an IDE platform is Eclipse.

In 2002 Eclipse.org[2] funded a research project to investigate and implement a BETA IDE plug-in for the Eclipse platform. First of all the plug-in should define a BETA mode in the Eclipse environment, but secondly the idea is to integrate all of the MjolnerTools separate tools into Eclipse.

[1] http://www.mjolner.com
[2] http://www.eclipse.org

7.2 The MjolnerTool Eclipse plug-in

As shown in the example in chapter 6 recognition of certain file extensions is easily accomplished. For creating a BETA IDE the default editor will not suffice so it is extended to satisfy the needs of a BETA editor. All of the other BETA IDE functionality is added as other Java classes. The initial BETA IDE implemented purely in Java has support for the following features:

- A BETA editor that activates on opening of `.bet` files
- Colouring of BETA keywords
- An outline view of the BETA code
- A BETA compiler with a console view, showing the compiler output
- Marking of compiler errors in the code
- Once compiled to an executable it is possible to run the code

The plug-in can be downloaded from http://daimi.au.dk/~beta/eclipse/ and on figure 7.1 a screen shot of the running MjolnerTool plug-in can be seen.

7.3 Interoperability

Eclipse plug-ins must be written in Java, so implementing a BETA IDE in Eclipse could be done in Java. But all of the fancy features wanted in a BETA IDE are already implemented in the MjolnerTool (written in BETA), and a full reimplementation in Java is not realistic. The ideal solution would be to somehow let the existing BETA code in MjolnerTool connect with the Java code in the Eclipse framework.

The challenge is therefore to allow interoperability between the Java and BETA languages, i.e. object creation and method invocation in BETA done from Java. If this can be done it is possible (with some adaptations) to reuse large parts of the old BETA code appearing in the MjolnerTool source.

7.4 The JNI Solution

The JDK library provides an interface to native methods. This native interface is called JNI (Java Native Interface) and allows code running in the Java Virtual Machine to

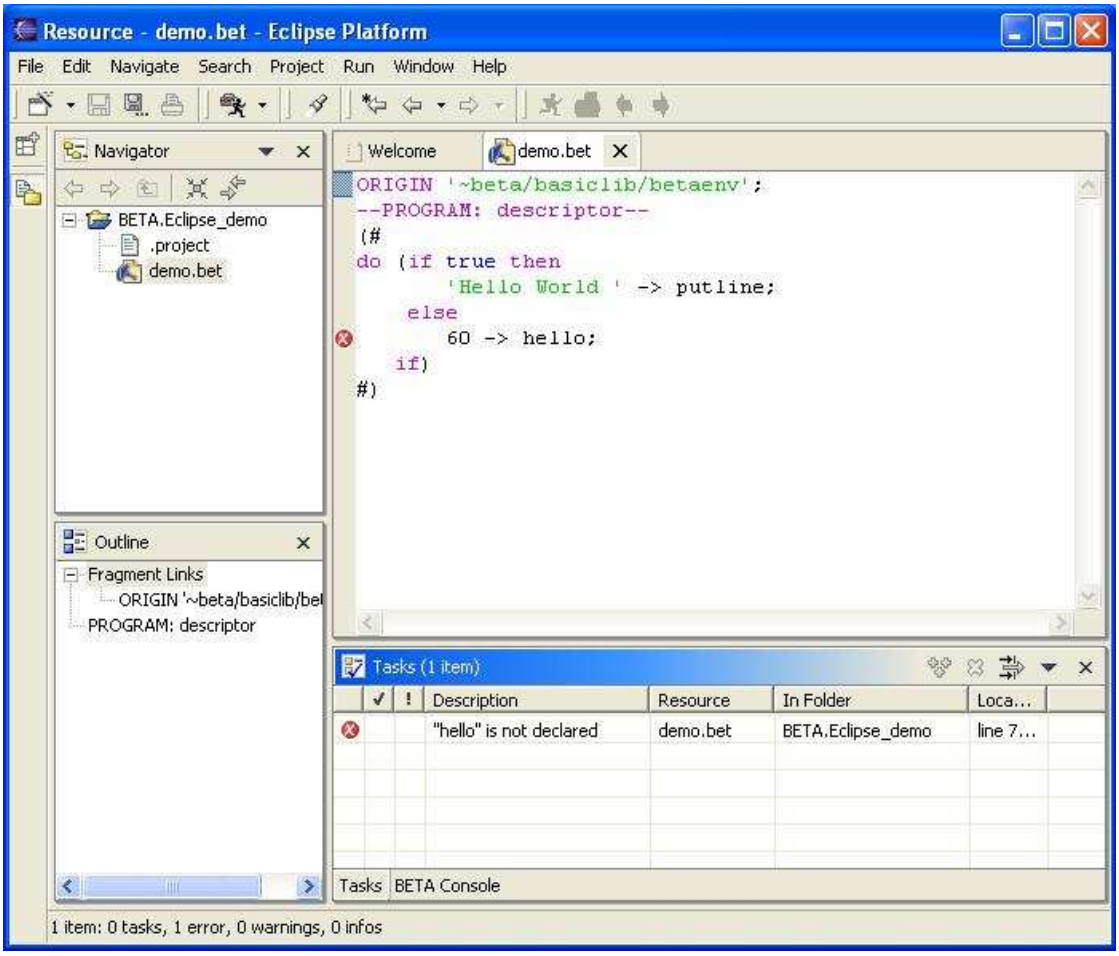

Figure 7.1: Screen shot of the MjolnerTool plug-in. The outline shows the fragments and origin/include parts, and a compile error is marked in the editor

access library methods written in C, C++ or assembly[3]. Since BETA has an interface to C it could be possible to access BETA from Java *via* C. This is achieved in a number of steps described below.

From BETA to C

The C interface from BETA allows C library functions to be called. When patterns in BETA have the super-pattern `external` the compiler will generate a call to an external C method having the same name as the pattern. Below there is an example of a declaration of a pattern (`getTime`) that inherits from the `external` pattern.

[3] http://java.sun.com/docs/books/tutorial/native1.1/concepts/index.html

```
getTime: external
  (# time: @integer
     exit time
  #);
```

An enter-part could also be provided, but a do-part cannot be specified since the function is external.

To make it work one need to specify where to find the C obj-file containing the method, allowing the BETA compiler to link with this. This can be done with the `OBJFILE` declaration as exampled below [4].

```
ORIGIN '~beta/basiclib/betaenv';
OBJFILE nti 'timelib.obj';
```

From C to BETA

The above was a description of how to get from BETA to C, but we were interested in the other direction. This is actually possible, although in a similar but more complicated way. The methods to be accessed from C, must still extend the super-pattern `external`, but now the methods need a do-part. The first imperative of the do-part must be `CExternalEntry` to tell the compiler to generate an entry point using C name and calling conventions and which creates and calls the actual BETA object when called. The call from C works as a callback which means that from the BETA side one has to install the function pointer before it can be used from C. A small example of the BETA-side of the code is given below.

```
(#
  setTime: external
    (# time,errorCode: @integer;
    enter time
    do CExternalEntry;
       time -> time.setTime -> errorCode;
    exit errorCode
    #);

  install_setTime: external (* an external which is hereby *)
    (# setTime: ##external  (* given a function pointer    *)
    enter setTime##         (* to the above BETA entry     *)
```

[4] See online BETA manual for reference: http://www.daimi.au.dk/$\sim$beta/doc/tutorial/tutorial-31.html

```
    #);
do
  setTime## -> install_setTime;
#)
```

The function `install_setTime` inherits from `external` (i.e. a call to a C function named install_setTime) and the C layer of the code is displayed below. The function `setTimeCB` is then a C method that can be called in C and it will call the BETA layer.

```
#include <stdio.h>

/* Type definition of the function pointers */
typedef int(*setTimePtr)(int);

/* Function pointers to save callback entry to Beta */
static setTimePtr setTime;

/* The callback install functions */
void install_setTime(setTimePtr stp) {
  setTime = stp;
}

/* The method invocation that uses the callback */
int setTimeCB(int i) {
  int j;
  j = setTime(i); // call BETA through the function pointer
  return j;
}
```

From Java to BETA

The problem of calling from Java to BETA is not solved with what is described above. To make C call BETA the initiative must be taken from BETA – BETA sends its function pointer to the C layer where the callback is installed and later used to call back through.

This problem is solved by extending the BETA native interface with the possibility of creating a library (on Windows platforms it is a `.dll` and for Sun platforms it is a `.so` file) of the BETA code[18]. This library can then be loaded from Java – loading of the library installs the callbacks – and the C methods can be called using JNI. From the C methods a further call to BETA is possible.

Using this approach, i.e. three programming languages and three compilers it is

possible to reuse some of the existing BETA code from the MjolnerTool as plug-in functionality in Eclipse.

Adding Indent Strategy

For Eclipse text editing, a so-called *indent strategy* can be associated. This specifies a default behaviour for indentation of code. A default indent strategy called `DefaultAutoIndentStrategy` exists in the Eclipse class library. Implementing a specific language indent strategy is obtained by overriding methods in this class.

In the Mjølner System library there is an implementation of an indent strategy for BETA code. The goal is to use this existing BETA library to implement the indent strategy in the editor plug-in.

The implementation – illustrated in figure 7.2 – consists of three layers, deployed into a Java part and two external libraries. The Java part implements the Eclipse extension point. The library entitled `JniIndentWrapper` contains all the "plumbing" needed to call C from Java whereas the library entitled `CwrappedBETALibrary` contains the BETA code to be called by Java as well as a set of C functions exposing the BETA library to JNI.

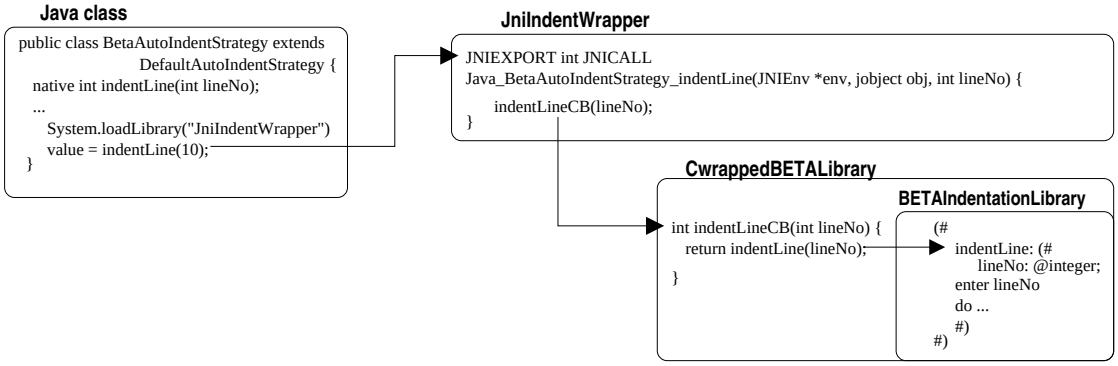

Figure 7.2: Java BETA interoperability using JNI

This form of interoperability requires the BETA plug-in to be written in Java, but allows some part of the functionality to be implemented in BETA. This facilitates reuse of existing BETA code. The code for the Java class calling the BETA libraries through JNI is sketched below:

```
class BetaAutoIndentStrategy extends DefaultAutoIndentStrategy {
  native int indentLine(int lineNo); // External method
```

```
  public BetaAutoIndentStrategy() {
    System.loadLibrary("JniIndentWrapper"); // Load external library
    ...
  }
  public void customizeDocumentCommand(IDocument d,
                                        DocumentCommand c) {
     ...
     value = indentLine(lineNo); // Call external method
     ...
  }
}
```

Evaluation

The implementation of the BETA indent strategy by reusing BETA libraries through JNI consists of more than 1000 lines of BETA code in several source files. This shows that this technique is possible to use for a non-trivial example. However, it has several disadvantages: one problem is that three different languages are used. This makes the integration tedious and complicated. Not only is it complicated to write the code, but more importantly once written and compiled it is far from certain that it will work. Loading of the libraries happens at run-time and this late binding of the code makes it impossible for the compiler to make even simple syntactic and semantic checks on e.g. the various method invocations. When a run-time error occurs in external code there is little information from the JVM. Often one has to suffice with the error message `Crashed somewhere outside the VM`. Debugging such a bug is done by carefully looking at the source code and using print statements everywhere!

Overall this solution is far from good. It solves the problem, but in a very inconvenient way for the implementor.

7.5 Java Bytecode Solution

The idea of rewriting MjolnerTool in Java was arguably unrealistic, but what about rewriting the BETA compiler? Indeed it is also an enormous task to change the backend of a compiler and let it generate different code, but luck has it that another research project porting the BETA compiler to Java bytecode is nearing its completion (the BETA compiler that generates Java bytecodes will be called JBeta further on).

With the JBeta compiler at hand it is possible to translate BETA source to Java class files containing Java bytecodes – thereby allowing for Java classes to extend BETA

patterns and vice versa.

Writing a plug-in with the JBeta compiler at hand makes things much nicer. JBeta makes it possible to reuse the MjolnerTool BETA code with only a few modifications.

Details of the Language Mapping

The mapping of the BETA language to Java bytecodes is challenging. BETA is more general than Java with the most profoundly different language constructs listed below.

- Class and method are united in the so called pattern
- General nesting of patterns
- `inner`-construction instead of `super`
- Genericity supported by use of virtual patterns
- Multiple return parameters
- Patterns have an enter-do-exit semantic

Consider the following pattern:

```
Calculator:
  (# R: @integer;
     set: (# V: @integer enter V do V -> R #);
     add: (# V: @integer enter V do R+V -> R exit R #);
  #);
```

The naive mapping to Java would be to let `Calculator` be a class with the two methods `set` and `add`:

```
class Calculator {
  int R;
  public void set(int V) { R = V; }
  public int add(int V) { R = R + V; return R }
}
```

This mapping is not good enough, because patterns `set` and `add` could be instantiated:

```
C: @Calculator;
A: ^C.add;
&C.add[] -> A[];
```

To capture this set and add are mapped to inner classes of Calculator:

```
class Claculator {
  ...
  class add {
    int add(int V) { R = R + V; return R}
  }
  ...
}
```

The actual mapping is more general in order to capture the `enter`, `do`, `exit` semantics of patterns. The details are not discussed here, instead see [1] for a complete mapping.

JVM Based Eclipse Integration

Above it was described how the indenter library written in BETA was accessed from Java in the plug-in using JNI. With JBeta this is much easier.

The Java class `BetaAutoIndentStrategy` can now be replaced with the BETA pattern inheriting from the Java class `DefaultAutoIndentStrategy`:

```
BetaAutoIndentStrategy: DefaultAutoIndentStrategy
  (# customizeDocumentCommand:
       (# d: ^IDocument; c: ^DocumentCommand; value: @integer;
       enter (d[], c[])
       do ... 10 -> indentLine -> value (* Call BETA *)
       #)
  #)
```

This pattern calls the indentation library directly, so with an implementation of BETA on JVM it is possible to drop all of the external libraries used for JNI. Note that the solution could just as well have kept `BetaAutoIndentStrategy` in Java, and accessed the BETA library from there. This is up to the programmer.

Some of the methods appearing in the class `BetaAutoIndentStrategy` take arguments of non basic type. To type check the access of instance variables or method invocation of

these externally defined objects JBeta has to know more about them. This can currently be done in the two following ways.

- Write the interface yourself. The code below shows how `IDocument` should look if one wants to use its methods.
- Instead of writing the interface oneself a compiler has been made. It compiles from Java bytecode to BETA source, and only generates the interface. The generated file is then simply included in the file referring the corresponding external class - for instance like `INCLUDE 'org/eclipse/jface/text/IDocument';`.

```
IDocument: externalClass
  (# get: proc
       (# result: ^String
       exit result[]
       #);
     getLineOfOffset: proc
       (# input, output: @integer;
       enter input
       exit output
       #);
     getLineOffset: proc
       (# input, output: @integer;
       enter input
       exit output
       #);
     replace: proc
       (# i1, i2: @integer; t : ^String;
       enter (i1, i2, t[])
       #);
  do '|org/eclipse/jface/text/IDocument' -> className;
  #);
```

Note that the declared interface contains a `do`-part. This specifies the full package name of the class. In this case the name contains a "|" at the beginning, which is used to signal the compiler that the current class is an interface, and therefore a special calling convention must be used.

Fate

The implementation of the indentation algorithm uses the regular expression BETA library. This library is implemented as an interface to a C library, and the porting of

this library to Java required much use of complicated JNI calls.

So the pure Java solution contains a native library, and is thereby not totally pure. But still more pure than the first solution.

Evaluation

Some advantages have already been mentioned, namely how elegant and easy it is to write a Java plug-in in BETA.

The disadvantage of this solution is that it is not easy to modify the compiler in this way. It is a task that requires time and careful thinking. A system as the BETA Mjolner system contains a lot of libraries, and there is a lot of work related to the porting of these libraries as well. And as seen in the previous section not all libraries consists of pure BETA code.

When referencing and using external classes from BETA the JBeta compiler has to do type checking on these external classes to ensure that the type rules are obeyed when calling methods on them. It was mentioned that this could be done by using a special compiler, that from a Java class file generated the BETA interface. So is this compiler like the IDL compiler to be used in CORBA? The answer is no. The IDL compiler in CORBA is used to generated all the infrastructure needed on both the client or server side of the application. The user then extends some of the generated classes and adds the wanted functionality for his application. In our case the interface compiler generates code to be used only at compile time. It is a question of making the JBeta compiler clever enough to do the type checking on class files. To underline the difference again: CORBA needs the IDL compiler to create the complicated infrastructure of the code needed for the application. JBeta has to understand the existing external code in order to do type checking, specifically the class and method signatures.

7.6 Summary

We have seen two different approaches to solving the problem of interoperability between two “components” that cannot communicate directly.

The first approach introduces a common language known to both components through which they can communicate. This closely resemblances what CORBA does and gives many layers of code. Layers that do nothing but dispatch method invocations. Even though the code of the different layers is simple there are still many layers, and confusing

errors arise easily.

The second approach uses a commonly known runtime environment, and lets both components generate code that runs in there. To compare this solution to known solutions this looks like a .NET solution. The common runtime environment allows the compiler to type check method invocations. This eliminates many runtime errors as opposed to the previous solution. But this, of course, requires a new compiler.

8 Integration with JDT Debugger

In the standard installation of Eclipse a Java IDE is contained. It consists of many different tools called Java Development Tools (JDT). JDT allows Java programmers to program, edit, test, debug and run Java applications.

The BETA IDE described so far gives the user the possibility to edit, indent, compile and run BETA programs. Another important part of an IDE is the debugger. As mentioned above, the MjolnerTool has a native debugger, so there are two solutions for adding a debugger to the BETA plug-in: use the classic BETA debugger or somehow use the Java debugger available in the Eclipse Java-IDE, called the JDT debugger.

Using the classic BETA debugger would require the following. First, the existing code should be refactored, structuring it so there is a clear separation of UI and non-UI code. Second, the non-UI part of the debugger would then have to be modified making it fit in the Eclipse launching/debugging framework – which means it would have to fulfil the contracts of the plug-ins by implementing the correct Java interfaces.

Using the JDT debugger is possible because it operates on Java Bytecodes, and that is what JBeta generates. So the idea is to compile the BETA source using JBeta and when debugging then do "whatever JDT does when debugging".

This chapter first describes how to use the BETA launcher and debugger in the BETA plug-in, and then the concepts needed of the launching framework in Eclipse are described and finally a description of the implementation details of the JDT debugger in the BETA plug-in.

8.1 Launching and Debugging BETA Applications

In figure 8.1 the two most important toolbar menus are shown – Run and Debug. The Run menu is concerned with launching the application and the Debug menu with debugging. The two menus look almost the same: Run contains menu entries `Run As` and `Run...` and Debug contains `Debug As` and `Debug...`. The interesting menu entries are the `Run...` and `Debug...` and initially they do the same thing, namely prompt the user

for configuring a launch configuration. Now, what is a launch configuration?

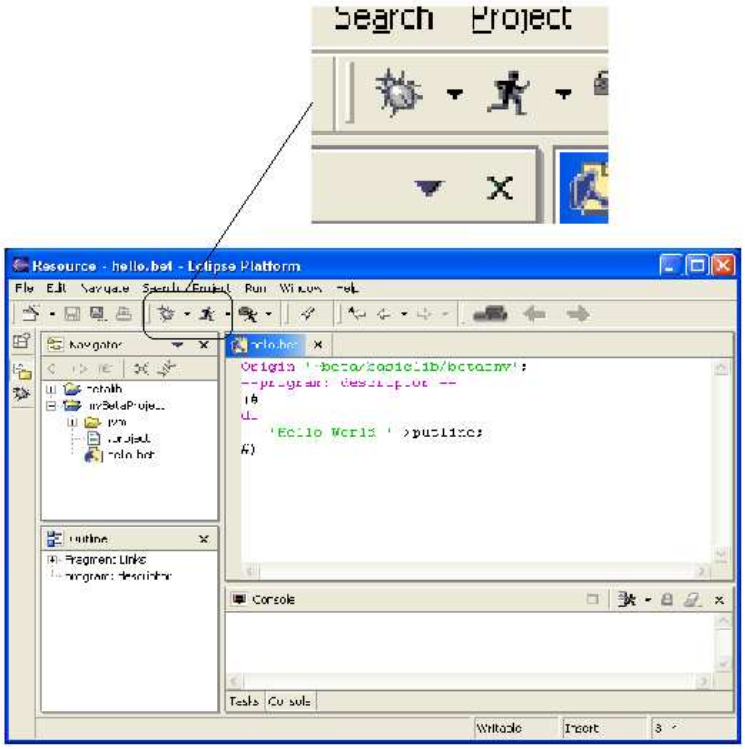

Figure 8.1: BETA IDE – the toolbar has debug and run menus

A launch configuration is an object describing the attributes needed by a specific application when it is to be launched. As an example consider launching a Java program. The launch configuration object would, for instance, hold information about the classpath needed by the application.

Clicking the `Run...` menu entry opens a window specific to creating a launch configuration. On figure 8.2 the window is shown. On the left side of the window one has to choose what kind of application there is to be launched. Of course, in this example, we choose "BETA Application" which opens the tab view seen on the right of the window. In this tab view there are three textfields: one to specify what file it is to be executed (here hello.bet), one to specify the main type (here beta.program) and an optional textfield where it is possible to specify additional arguments to the JVM. Lastly there is a name textfield at the top of the window that allows the user to specify a name for this instance of the launch configuration object – here we specify it as "Beta Program Hello".

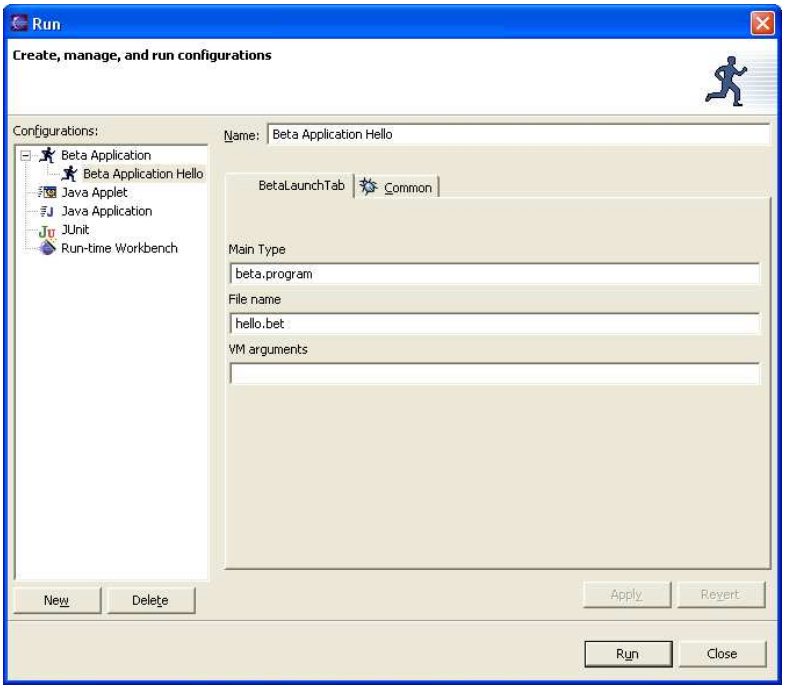

Figure 8.2: The Beta launch configuration tab

With all this there has now been created a launch configuration object with the name "Beta Program Hello" connected to the source file hello.bet and with main type beta.program. So this configuration has all the information needed to launch `hello.bet`.

With the creation of the launch configuration object the run and debug menus have both dynamically expanded. They now also contain an entry corresponding to the specific launch configuration (the "Beta Program Hello" in figure 8.3). The launch configuration object is persisted so that it will be remembered even if Eclipse is shut down.

Launching

The setup has now been done, i.e. a launch configuration has been made, so now we're ready to launch the "hello world" application.

This is now done by selecting the "Beta Program Hello" menu item in the run menu (see figure 8.3). A screen shot of Eclipse after having done that can be seen on figure 8.4.

Figure 8.3: The menu has been expanded with the highlighted menu entrance `Beta Application Hello`

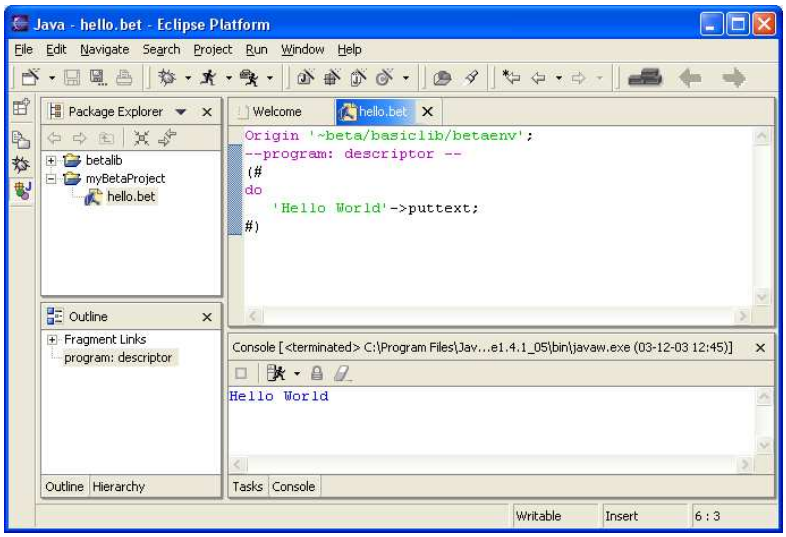

Figure 8.4: Eclipse after launch of "Hello World" program. Notice the output in the console

Debugging

Instead of using the `Run` icon from the toolbar the `Debug` icon launches the BETA code in debug mode. The launch configuration defined before ("Beta Program Hello") has also appeared in the debug menu, so choosing that launches the application in debug mode. If breakpoints are defined the execution will stop when the corresponding lines of code are reached.

Breakpoints can be defined by double clicking the vertical ruler on the left side of the editor. When a breakpoint is set a little icon is shown to indicate that a breakpoint is here. Double clicking on an existing breakpoint icon removes it.

On figure 8.5 a debug launch is shown. Notice that in the upper right corner of the window the visible objects are shown. When the execution reaches the breakpoint the value of the variable i can in fact be changed. This change will then apply when the

execution continues.

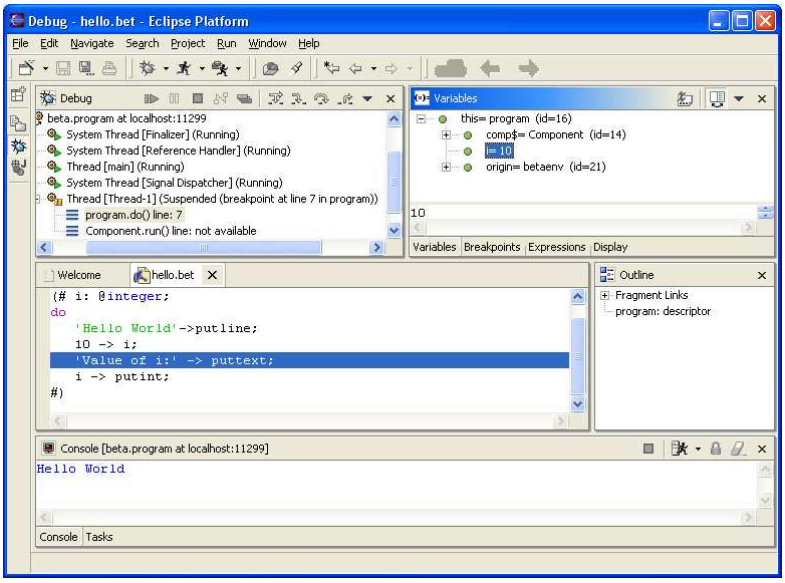

Figure 8.5: Debugging Beta code

8.2 The Launching Framework in Eclipse

The launching framework of Eclipse is very general and therefore has many declared extension points and interfaces to use when writing both launching and debugging actions to your plug-in[1]. Since the launching/debugging added to the BETA IDE is closely connected to the JDTs not many extension points and interfaces need to be implemented. The two concepts of the Eclipse launching framework that have to be used are *launch configurations* and *launch configuration types*.

Launch Configurations

We have already encountered a launch configuration. In the description of how to use the BETA launcher we saw how to create a launch configuration. This launch configuration holds the information of the given program, i.e. it knows everything that has to be known when launching one specific application. A launch configuration does not do anything, but has information.

The interface for a launch configuration is `org.eclipse.debug.core.ILaunchCon-`

figuration, but clients are not supposed to implement this interface. Instances of this interface are provided by the framework via the tab view window we looked at in the previous section (figure 8.2).

Launch Configuration Types

A launch configuration type performs the actual launch, and it is therefore more complicated than a launch configuration. One declares a launch configuration type in the plug-in manifest, and an example is written below.

```
<extension
 point="org.eclipse.debug.core.launchConfigurationTypes">
 <launchConfigurationType
    id="UniqueId"
    delegate="org.eclipse.ExampleLaunchConfigurationDelegate"
    modes="run,debug"
    name="Example Application">
  </launchConfigurationType>
 </extension>
```

In this extension point one declares first of all `name` and `id`. The `modes` can be either `run`, `debug` or both and is used distinguish the two possible kinds of launch modes.

The important attribute is the `delegate`. This is the fully qualified identifier for a Java class implementing the interface `org.eclipse.debug.core.ILaunchConfiguration-Delegate`. This interface contains only one method, namely:

```
public void launch(ILaunchConfiguration config,
                   String mode,
                   ILaunch launch,
                   IProgressMonitor monitor) throws CoreException
```

The task for this method is simply to launch the application, i.e. extract the information defined by the method arguments and put it together in a string to use as argument to `System.exec()`.

config : This `ILaunchConfiguration` object contains all the attributes needed to launch the application.

mode : The argument `mode` is a string that says either `run` or `debug`. Depending on what it is the argument to `System.exec()` changes. Example: when launching a Java application in "Run"-mode on Windows, the argument is:

```
 C:\Java\j2re\Bin\javaw.exe -classpath
c:\eclipse\workspace\Example Main
```

Whereas in ”debug”-mode it is:

```
C:\Java\j2re\Bin\javaw.exe -classpath
 c:\eclipse\workspace\Example
 -Xdebug -Xnoagent -Djava.compiler=NONE
 -Xrunjdwp:transport=dt_socket,suspend=y,
 address=localhost:9458 Main
```

launch : The `ILaunch` argument represents the launch, and all processes created by the delegate should be added to this object.

monitor : Lastly the `IProgressMonitor` is a UI obejct, used to indicate the completion percentage of the operation. If the user clicks cancel the running processes should be terminated.

The JDT Launch Configuration Type

The JDT debugger is large and complex and it is pointless to go through all of its implementation details. Instead we will look at what concerns us, namely the launch configuration type it uses when launching local Java applications. This configuration type is defined as follows:

```
<extension
  point = "org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
      id="org.eclipse.jdt.launching.localJavaApplication"
      name="%localJavaApplication"
      delegate="org.eclipse.jdt.internal.launching.
                JavaLocalApplicationLaunchConfigurationDelegate"
      modes= "run, debug">
  </launchConfigurationType>
</extension>
```

So the class JavaLocalApplicationLaunchConfigurationDelegate does whatever has to be done when Eclipse launches a Java application.

8.3 Creating a BETA Launch/Debug Mode

Now we have all the pieces needed to create a launch configuration type for BETA applications. We need to define a delegate class for launching and debugging BETA applications. But what is exactly the difference from the delegate used to launch local Java applications and the one we need? When compiling BETA files using JBeta, a jar file is created. Launching the do-part of the program descriptor for a program called `hello.bet`, can be done from a shell with the following command line `java -classpath jvm/hello.jar beta.program`. So it is just a matter of setting up the special classpath and use `beta.program` as the main type. This means that the extension point in the BETA launch uses exactly the same delegate as JDT.

The extension added to our `plugin.xml` looks as follows:

```
<extension
   point="org.eclipse.debug.core.launchConfigurationTypes">
   <launchConfigurationType
      name="Beta Application"
      delegate="org.eclipse.jdt.internal.launching.
                JavaLocalApplicationLaunchConfigurationDelegate"
      modes="run, debug"
      id="daimi.launching.BetaconfigurationType">
   </launchConfigurationType>
</extension>
```

UI

The UI related to the debugger is an important part of the debugger because it dictates how to interact with it.

In the Eclipse debugging UI framework the most important piece is the launch configuration tab group. A launch configuration tab group contains launch configuration tabs, and through these tabs you have the ability to create, initialise, edit and store launch configurations – recall that these were the very important objects that held the information on how to launch a certain application. But before coming to the actual tab we first have to look at how to create the launch configuration tab group.

As always we have to consult the plug-in manifest, so the following extension is added to our `plugin.xml`:

```
<extension
```

```
  point="org.eclipse.debug.ui.launchConfigurationTabGroups">
  <launchConfigurationTabGroup
    type="daimi.launching.BetaconfigurationType"
    class="daimi.launching.ui.BetaTabGroup"
    id="daimi.launcing.ui.BetaTabGroup">
  </launchConfigurationTabGroup>
</extension>
```

The two important attributes to mention are `type` and `class`. The `class` attribute is a pointer to the class file that implements the interface `org.eclipse.debug.ui.ILaunchConfigurationTabGroup`. The `type` attribute refers to what launch configuration type this UI widget is connected to. So the contents of this type attribute is the same as the `id` of the BETA launch configuration type declared in the previous section.

A launch configuration tab group is just a manager of the launch configuration tabs it contains. So the interesting thing is not the tab group itself, but the tabs it creates. For the BETA launcher there is only one tab needed – in fact we have already encountered it (shown in figure 8.2). In this tab there was specified the name of the file that were to be launched and its main type name. To understand what these are used for in the launch configuration we have to look at what JBeta does when generating Java bytecode.

Using the JBeta compiler on a BETA file, lets say `hello.bet`, a sub directory called `jvm` is created. This directory contains the jar file `hello.jar`. The `public static void main()` method in BETA is the do-part of the `program descriptor`. This do-part is generated to the class file `program` in package `beta`. To run the program in a console the command line should be `java -classpath jvm/hello.jar beta.program`.

So given the file name it is possible to generate the classpath and the do-part of the `program descriptor` is always `beta.program`.

BETA Launch Configuration Tab

The BETA launch configuration tab is created by implementing the interface `ILaunchConfigurationTab`. The three most important methods on this interface is:

```
public void setDefaults(ILaunchConfiguration)
public void performApply(ILaunchConfiguration)
public void initializeFrom(ILaunchConfiguration)
```

The `setDefaults` method is called when a new launch configuration is created. This happens when the user presses the `new` button (figure 8.2). In this method the pro-

grammer has to somehow make a qualified guess of what the values of the configuration should be. This should be possible because, as seen before, given the name of the file it is easy to calculate the classpath, and the main type is always `beta.program`.

The `performApply` method is called whenever the user presses the `apply` button in the tab. It copies the values from the textfields into the launch configuration. In our case the classpath and the main type has to be set in the launch configuration.

`initializeFrom` does the opposite of `performApply` – it copies values from the launch configuration to the textfields of the tab.

The launch configuration tab holds the information needed when BETA applications are launched.

8.4 Breakpoints

As explained above, launching a class file generated from a BETA source is not a problem. But what about the next step – creating a breakpoint and debugging the application?

Breakpoints are set by the programmer on source code level. The execution of the application should then stop when the executing code corresponds to the annotated source line. So breakpoints are *connected* to source files and since Java and BETA sources are different the possibility of reusing JDT classes directly is smaller.

The Classfile Format

Fortunately it is possible to add some additional information in class files. It is possible to write line number information and specify what source file the classfile is generated from.

The example below shows the source file of a simple BETA program with line numbers added:

```
1  ORIGIN '~beta/basiclib/betaenv';
2  --PROGRAM: descriptor--
3  (#
4  do
5     'Hello, ' -> puttext;
6     'World!'-> puttext;
7  #)
```

The result of compiling this with JBeta is a jar file containing many class files. The code corresponding to the `do` part of the program is located in the file `program.class`. Using the GNU[1] tool `jcf-dump` on this class file it is made readable. The relevant parts of this dump is shown below:

```
Method name:"Do" public Signature: 17=()void
Attribute "Code", length:121, max_stack:100, max_locals:100, code_length:71
  0: aload_0
  1: getfield #21=<Field beta.program.origin beta.betaenv>
  4: ldc #33=<String "Hello, ">
     ...
 35: aload_0
 36: getfield #21=<Field beta.program.origin beta.betaenv>
 39: ldc #57=<String "World!">
     ...
 67: invokevirtual #55=<Method beta.betaenv.puttext (beta.text)void>
 70: return
Attribute "LocalVariableTable", length:12, count: 1
  slot#0: name: 14=this, type: 15=beta.program (pc: 0 length: 70)
Attribute "LineNumberTable", length:14, count: 3
  line: 5 at pc: 0
  line: 6 at pc: 35
  line: 1 at pc: 70
...
Attribute "SourceFile", length:2, #2="hello.bet"
```

The two lines of code, from the source file (`'Hello, '-> puttext` and `'World!' -> puttext`), are referenced at the bottom of method do. It can be seen that line 5 is encountered at program counter (pc) 0 and line 6 at pc 35. At the end of the dump the attribute `SourceFile` specifies that the source file this class file is generated from is `hello.bet`.

With the possibility of adding this information to class files, it is possible to create generic Java debuggers – all information needed is available in the class file.

The JDT Debugger

The Eclipse class `JDIDebugModel`[2] is an implementation of the Java Debug Interface (JDI) in SUN Microsystems' Java Platform Debugger Architecture (JPDA)[3]. It contains

[1] See http://www.gnu.org/

[2] located in package `org.eclipse.jdt.debug.core`

[3] See http://java.sun.com/j2se/1.4.2/docs/guide/jpda

static methods for adding breakpoints to the debug model. For the purpose of debugging BETA, the interesting method is `createLineBreakpoint`. It has the following signature:

```
createLineBreakpoint(IResource resource,
                     String typename,
                     int lineNumber,
                     int charStart,
                     int charEnd,
                     int hitCount,
                     boolean resister,
                     Map attributes)
```

The below description of the arguments to this method is taken from the Eclipse online documentation[4].

resource : the resource on which to create the associated breakpoint marker.

typeName : the fully qualified name of the type the breakpoint is to be installed in. If the breakpoint is to be installed in an inner type it is sufficient to provide the name of the top level enclosing type. If an inner class name is specified, it should be formatted as the associated class file name (i.e. with $). For example, example.SomeClass$InnerType, could be specified, but example.SomeClass is sufficient.

lineNumber : the lineNumber on which the breakpoint is set - line numbers are 1 based, associated with the source file in which the breakpoint is set

charStart : the first character index associated with the breakpoint, or -1 if unspecified, in the source file in which the breakpoint is set

charEnd : the last character index associated with the breakpoint, or -1 if unspecified, in the source file in which the breakpoint is set

hitCount : the number of times the breakpoint will be hit before suspending execution - 0 if it should always suspend

register : whether to add this breakpoint to the breakpoint manager

attributes : a map of client defined attributes that should be assigned to the underlying breakpoint marker on creation, or null if none.

[4] http://help.eclipse.org/help21/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/debug/core/JDIDebugModel.html

What complicates the setting of breakpoints in BETA sources, is the *type name* that has to be specified. As described in section 7.5, a BETA source file is compiled into several `class` files, i.e. many types. Given a specific BETA file and a line number it is not trivial to find the corresponding generated `class` file. In a future version, the BETA compiler would, however, be able to supply this information for the BETA plug-in [5].

The Eclipse debugging model could be more friendly for BETA and the numerous other languages targeting JVM[6] by using a less strict coupling to JPDA.

Since Java class files contain information about source file names and line numbers for the bytecodes, the debugger should be able to find the break locations given just the source file name and line number: For classes already loaded, the debugger could check if any of these corresponds to the given source file, and set the breakpoint. If no such class is currently loaded the debugger could defer the breakpoint setting and repeat the analysis whenever a new class is loaded.

In the proof of concept implementation of the BETA IDE it is possible to add *line breakpoints* to the BETA source[7]. It is, however, impressive that it is possible to develop generic tools like Eclipse and JVM that given a compiler that generates JVM bytecode can support a full symbolic debugger for a considerably different language like BETA.

Setting Breakpoints in BETA Mode

Breakpoints are added to the BETA source by double clicking in the left vertical ruler. This is implemented by adding an extension to the `ui.editorActions` extention point with the attribute `actionID` saying `RulerDoubleClick` as can be seen below.

```
<extension point="org.eclipse.ui.editorActions">
 <editorContribution
   <action
     label="Beta Breakpoint"
     class="daimi.launching.BetaBreakpointRulerActionDelegate"
     actionID="RulerDoubleClick"
     id="betaeditor.actions.BetaBreakpointAction">
   </action>
```

[5] The activation of JBeta when building BETA projects is accomplished by calling an external `jbeta` executable

[6] Programming Languages for the Java Virtual Machine: http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[7] At the time of writing it is only possible to break inside the `PROGRAM` fragment, because we have not yet integrated the above mentioned program analysis to determine the class name from a source file position

```
 </editorContribution>
</extension>
```

When the user double clicks the ruler the class pointed to by the `class` attribute sets a breakpoint.

Debugging BETA in Visual Studio .NET

Investigations of using Microsoft Visual Studio .NET [14] (VS) as an IDE supporting BETA has also been done[1]. The integration requires BETA to be ported to Microsofts CLR/.NET platform and the integration is somewhat similar to the Eclipse integration.

To obtain a language integration like what has been implemented for BETA in Eclipse, i.e. syntax colouring, control of the BETA compiler etc., hard work is needed: like in Eclipse the VS IDE is programmable and extensible. The APIs for doing this consists of a large number of COM interfaces that need to be implemented to obtain the integration. Thus, it requires a lot of non-.NET work to do the integration. However, a beta-release of a so-called *managed VSIP API* has been announced from Microsoft [15]. Using this it is possible to write the integration like it is done in Eclipse – write the code in BETA and compile it to .NET bytecodes [2].

Writing a language IDE in VS is similar to doing it in Eclipse, but VS is somewhat better prepared when it comes to debugging. The multi-language of .NET support manifests itself by the ease of getting a BETA program into VS and do almost complete source level debugging on it. To make VS do this all that is needed is to include source file and line number information into the generated byte code for .NET/CLR, and then just open an executable generated from BETA in VS. This immediately allows VS to identify the source files, do source level stepping, insert break points, inspect objects and even change values of variables. Note that no changes to VS was needed to accomplish this.

8.5 Summary

Using the JBeta compiler it is possible to reuse the JDT launcher. The JDT debugger, however, is more complicated. Even though it would be adequate to state line number and file name to the JDT debug model when adding a breakpoint, the JDT debug model insists on knowing what enclosing type the breakpoint is added in – this is because the JDT implementation of a Java debugger follows a Sun Microsystem standard (JDI). For

BETA (and probably many other languages as well) it would be much nicer if breakpoints were specified by only line number and file name.

In VS debugging is implemented more generically which makes the use of it easier than reusing the JDT debugger. The JDT debugger is an implementation of a Java debugger specification made by Sun Microsystems, and was probably not designed to debug other languages than Java.

9 Evaluation

This section concludes the second part of this thesis with comments and discussions of the author's experiences in using Eclipse as a programming tool for developing support for the BETA language in Eclipse.

9.1 Eclipse Community

The development of Eclipse is ongoing, and at the time of writing version 3.0 is soon on the way. Besides from the actual releases it is possible to download and install nightly builds and last stable builds from the Eclipse.org home page.

This home page supports a whole forum of news groups, addressing various topics concerning plug-in development of specific tools (e.g. C++ development tools) or tools in general[1]. In connection with the various projects it is even possible to access their source code from the public Eclipse CVS repository, and this has turned out to be very helpful for the development of the BETA IDE – some aspects of the Java launching implementation was directly usable for BETA. The combination of newsgroups and CVS repositories breeds a mutually advantageous situation for both the Eclipse developers and plug-in writers.

Many Eclipse related events are constantly under way. The **eTX** (Eclipse Technology Exchange) is a workshop usually held in connection with larger conferences such as OOPSLA[2] and ETAPS[3] where developers meet and share experiences related to Eclipse development both practical and academical.

Overall Eclipse is a promising effort in providing a general IDE platform.

[1] see http://www.eclipse.org/newsgroups/ for a list of news groups
[2] See http://www.oopsla.org
[3] Short for European Joint Conference on Theory And Practice Of Software

9.2 Plug-in Development

The Eclipse platform is designed to be a general IDE platform. It should be possible to support IDE plug-ins for every language. These language IDEs must be written in Java, but as illustrated in the two previous chapters, given a compiler emitting Java bytecodes allows for plug-ins to be written in virtually any language. But other strategies for developing plug-ins in foreign languages is not supported by Eclipse – of course there is JNI, but that is strictly a Java extension.

Documentation

As mentioned earlier, the Eclipse platform consists of a number of APIs. Therefore a great amount of documentation is needed, and also available[4]. Plug-in creation consists of two parts, namely a manifest and actual code. This separation is reflected in the layout of the documentation. Besides from general "getting started" and overall descriptions of the architecture the documentation for the plug-in developer is gathered in a folder called "Platform Plug-in Developer Guide". This guide has two important parts: an extension point reference (a complete list of all extension points of the Eclipse architecture) and an API reference (JavaDoc for all classes available in the APIs). In the extension point reference for every extension point an example of its usage is given along with useful comments on the meaning of the different attribute names. In the API reference, the JavaDoc of the different classes also have a brief description of how the class interconnects and should be instantiated/extended by the programmer. This description is not always fully understandable because sometimes it must be part of a whole collection of classes to make sense. For that some examples of plug-ins are given – e.g. examples on how to create new editors. These examples show the developer how to create some simple plug-ins. More useful is the Eclipse articles available from the Eclipse web site. Each article is a detailed description of a very specific area in the plug-in development. From the author's point of view it is always the first place to look when new areas of Eclipse has to be uncovered.

Evolving Eclipse

Since the development of Eclipse is ongoing, changes to the API occur. This, of course, affects the plug-ins using the modified APIs rendering them to no longer connect to the framework. The changes are (at least to the author's knowledge) for the better, and the

[4]See web page http://help.eclipse.org/21/documentation/index.jsp

news-group and documentation support when adhering to these changes is comprehensive.

Eclipse being open source is a great feature for the plug-in writer. All of the JDT is available as source which provides the developers with a enormous (probably the largest) sample of plug-ins. JDT being enormous means that the plug-ins defined by it is too large to actually be a real help. But sometimes one can extrapolate useful information to be mimicked in self made plug-ins. The general picture has been that JDT is too complicated to be of any help.

9.3 MjolnerTool

The implementation of the BETA IDE was done mainly be reusing existing tools from MjolnerTool. As mentioned MjolnerTool has been under development for many years and was first available as stand alone applications, but later merged. The code is spread out in different directories, each corresponding to one of the earlier stand alone tools.

The main problem is the way the MjolnerTool code is structured. For instance the indenter algorithm was implemented as a specialised `textfield` (a BETA graphical widget). The indenter algorithm was then directly inserted in the MjolnerTool by letting it use the indenter `textfield` instead of a normal `textfield`. Even though both the UI and non-UI code was located in the textfield the code was not particularly ugly. This is because of the BETA fragment system that allows for separating method implementation into different files (the `BODY` construct) – the code was nice and simple, but useless if it were to be used in another graphical widget.

Implementing an indent strategy in the Eclipse BETA IDE, was done by refactoring the indenter code – with the core indentation algorithm separated completely from the widget code.

Integrating the graphical tools of the MjolnerTool into Eclipse (the interface builder, UML diagram editor, POSTwimp interface builder, etc.) has not been done, but is expected to require a considerable amount of work,because the graphical system of BETA is not ported to SWT. So even though plug-ins can be written in BETA just as well as Java reusing tools existing tools from MjolnerTool is not straight forward.

10 Conclusion

Language interoperability is the ability for different languages to interoperate. To make it useful for the programmer certain demands have been defined.

The technologies studied have been evaluated with respect to these demands. The problem for CORBA is that it is designed with emphasis on distribution, and this makes it "heavy" to work with when only addressing language interoperabilty. At least two programming languages are needed to make it work, and also two compilers. Cross language inheritance is not possible, so the "first class citizen" demand is not fulfilled. The bottom line is that CORBA does not satisfy the interoperability demands.

Microsoft COM is a better technology with respect to the demands. Only one language and one compiler is needed when creating and interoperating with components. But the "first class citizen" fails when it comes to implementational inheritance. Programming paradigms containment and aggregation are suggested to allow for simulation of inheritance, but they are very complicated to implement.

The virtual machines JVM and CLR did satisfy all of the suggested demands for ideal language interoperability. Targeting these from different languages makes the language barriers disappear. With respect to object oriented languages, the two platforms are very similar, but when porting imperative or functional languages CLR is superior.

The IDE platform Eclipse was studied with focus on how to integrate existing tools. Eclipse is written in Java and for executing plug-ins uses JVM. The case study was on how to integrate MjolnerTool in Eclipse using language interoperability rather than reimplementation.

CLR being a better language platform than JVM could suggest IDE platforms to use this rather than JVM. The advantages for choosing JVM is that Java and JVM is widespread, available on every platform – and one of the design goals of Eclipse is availability on many platforms.

For the integration JNI could be used to bridge BETA with Java through C. This solution failed to meet any of the interoperability demands: foreign objects was accessed through two interfaces written in three different languages compiled with three compilers

and type safety could not be guaranteed. Indeed this solution was very unpractical.

Targeting JVM from BETA with the JBeta compiler allowed for a tight integration into Eclipse. This solution met all of the interoperability demands, and allowed for plug-ins to be implemented in BETA. With JBeta reuse of MjolnerTool was possible. With structural changes to the MjolnerTool code it was possible to integrate the indentation algorithm, as written in BETA, into the BETA IDE in Eclipse.

As JVM is a language platform generic tools is a possibility, and the Eclipse Java debugger was shown to be usable on BETA sources. Unfortunately, the implementation of the Eclipse debugger followed a Sun Microsystems specification of a Java debugger. When adding a breakpoint the class file containing the bytecodes for the corresponding source code has to be specified. This may seem fine for a Java source debugger, but with BETA it is not trivial to find this class file. The fact is that breakpoints could be added by only specifying source file name and line number – which would be trivial for all languages.

The closing remark is that even though we have JBeta and can reuse the debugger from Eclipse, hard work is still needed when porting the rest of MjolnerTool to Eclipse. Perhaps future frameworks will allow for an even easier integration – time will tell.

Bibliography

[1] P. Andersen, O.L. Madsen: *Implementing BETA on Java Virtual Machine and .NET – an exercise in language interoperability*, Computer Science Department, Aarhus University, January 2004.

[2] O.L. Madsen, P. Andersen, M. B. Enevoldsen: *Integration of BETA in Eclipse – an exercise in language interoperability*, Computer Science Department, Aarhus University, January 2004

[3] Arnold, Gosling, Holmes: *The Java Programming Language*, Third Edition, Addison-Wesley, Boston 2000, ISBN: 0201704331

[4] Dale Rogerson: *Inside COM*, Microsoft Press, February 1997, ISBN: 1572313498

[5] G. Coulouris, J. Dollimore, T. Kindberg: *Distributed Systems*, Third Edition, Addison Wesley ISBN 0201619180

[6] *Common Object Request Broker Architecture: Core Specification*, December 2002, http://www.omg.org/docs/formal/02-12-02.pdf

[7] Microsoft .NET: http://www.microsoft.com/net

[8] Microsoft .NET Common Language Runtime: http://msdn.microsoft.com/library/en-us/cpguide/html/cpconCommonLanguageRuntimeOverview.asp

[9] E. Meijer, J. Gough: *Technical Overview of the Common Lanugage Runtime*, 2001

[10] K. J. Gough, *Stacking them up: a Comparison of Virtual Machines, 2001* *http://sky.fit.qut.edu.au/~gough/VirtualMachines.ps*

[11] J. Singer, *JVM Versus CLR: A Comparative Study, 2003*

[12] The Mjølner System, Mjølner Informatics A/S: http://www.mjolner.dk/mjolner-system

[13] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*. ACM Press/Addison Wesley, Addison-Wesley, June 1993, ISBN 0-201-62430-3, 350 pages. Out of print – a copy can be downloaded from http://www.mjolner.com/mjolner-system/books_en.php

[14] Microsoft Visual Studio Home Page: http://msdn.microsoft.com/vstudio

[15] Microsoft Visual Studio Industrial Partners (formerly: Microsoft Visual Studio Integration Program): http://www.vsipdev.com

[16] J. Szurszewski: *We Have Lift-off: The Launching Framework in Eclipse*, 2003
http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

[17] A. Bolour: *Notes on the Eclipse Plug-in Architecture*, 2003
http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[18] P. Andersen: *DLL usage in BETA*, 2002
http://www.daimi.au.dk/~beta/eclipse/DLL.html

[19] O. L. Madsen: *COM Support in BETA*, 1999

[20] N. Lauritzen: *ALGEBRA I. Numbers, Relations and Groups*, 1999

[21] L. A. Stein: *Delegation Is Inheritance*, Brown University, 1987