Scripting with Java™
An evaluation of scripting with pre built Java applications

Daniel Pettersson
Scripting with Java™

An evaluation of scripting with pre built Java applications

Daniel Pettersson
Preface
This master’s thesis is the final assignment and the end of my education at Luleå University of Technology.

Bo Sehlberg and Frode Randers at Tietoenator Public & Healthcare AB gave the work to me and I like to thank them both for this interesting assignment, and at the same time give thanks to my examiner at Luleå University of Technology Kåre Synnes.

This thesis has the purpose of investigating how scripting will function together with pre built Java applications. It compares scripting with other methods like normal configuration management method and enterprise application integration. The goal is to solve problems like adding, removing and integrating functionality without a great development cost. My hope is that this work will give a clear idea on how scripting can be used to solve these problems.
Abstract
Tietoenator Public & Healthcare AB has a close cooperation with its clients. They offer solutions that make their daily work more efficient and have therefore developed different applications to their needs.

The company has discovered that old applications can easily be reused in some way to new customer needs. A method that group functionality into modules has been used to solve this problem. These modules are then reused and modified to fit a new customer need. The only problem with this method is that it can crave great knowledge about all modules in an application. This will crave a great investigation cost before one can add, remove, exchange or integrate functionality even if all functionality exists.

Normal Configuration Management Method (NCMM) is a method for handling situations like the ones described above. It states rules for building and making modules reusable to different situations. Another method can also be used that are called Enterprice Application Integration (EAI). This method is somewhat different compared to NCMM. It mainly focuses on making application interoperable with each other. Scripting can offer all things that NCMM and EAI do. This mainly depends on the fact that scripting is evaluated in clear text and offers modular reusable solutions. The report will therefore explore the usage of different scripting methods for minimizing this development/modification cost, and compare it to both NCMM and EAI.

The result of the evaluation shows that scripting is a good method to temporary solutions, i.e. to solutions that are not likely used for future development.
Table of contents

1. Introduction .......................................................................................................................... 7
  1.1 Background and purpose ................................................................................................. 7

2. Defining the work ................................................................................................................. 8
  2.1 Scripting tools and report limitations ............................................................................. 8
  2.2 Other methods that scripting are evaluated against ....................................................... 8
  2.3 Test program .................................................................................................................... 9

3. Definitions and concepts .................................................................................................... 10

4. Normal Configuration Management Method.................................................................. 12
  4.1 Application life cycle ..................................................................................................... 12
  4.2 Software Configuration Identifications .......................................................................... 13
  4.3 The usage of SCI to solve the defined test criteria......................................................... 14
    4.3.1 Add or remove functionality from an application ................................................... 14
    4.3.2 Integrating pre built applications............................................................................. 14
    4.3.3 Exchange functionality or a GUI ............................................................................ 15
    4.3.4 Customer support and installation aid..................................................................... 15

5. Middleware and Enterprise Application Integration (EAI) .......................................... 16
  5.1 Middleware..................................................................................................................... 16
  5.2 EAI .................................................................................................................................. 17
    5.2.1 Data-level EAI ......................................................................................................... 18
    5.2.2 Application interface-level EAI .............................................................................. 18
    5.2.3 Method-level EAI .................................................................................................... 20
  5.3 The usage of EAI to solve the defined test criteria ........................................................ 21
    5.3.1 Add, remove or exchange functionality from an application.................................. 21
    5.3.2 Integrating pre built applications............................................................................. 21
    5.3.3 Exchange a GUI ...................................................................................................... 22
    5.3.4 Customer support and installation aid..................................................................... 22

6. Scripting tools ..................................................................................................................... 23
  6.1 Tcl................................................................................................................................... 23
    6.1.1 Description of Tcl/Java ........................................................................................... 23
    6.1.2 Usage of tclBlend and jacl ...................................................................................... 23
  6.2 JavaScript ....................................................................................................................... 24
    6.2.1 Description of Rhino ............................................................................................... 24
    6.2.2 Usage of Rhino........................................................................................................ 25
  6.3 Description of Visual Basic and Java............................................................................. 25
  6.4 Description of Php.......................................................................................................... 26

7. The effects of scripting with an application ..................................................................... 27
  7.1 Do scripting break the Java object structure ................................................................. 27
  7.2 The effect of unauthorised editing of script-files ........................................................... 28
  7.3 The long term affect with scripting ................................................................................ 28
  7.4 Scripting and program security ...................................................................................... 29
8. Evaluation of the found scripting tools ................................................................. 30
  8.1 Evaluation of integrating applications................................................................. 30
    8.1.1 Methods ........................................................................................................ 30
    8.1.2 Script evaluation with the test application ...................................................... 31
    8.1.3 Conclusion for integrating applications ......................................................... 31
  8.2 Evaluation of adding functionality ................................................................. 32
    8.2.1 Methods ........................................................................................................ 32
    8.2.2 The test .......................................................................................................... 33
    8.2.3 Conclusion for adding functionality ............................................................ 33
  8.3 Evaluation of removing or exchanging functionality .................................... 34
    8.3.1 Methods ........................................................................................................ 34
    8.3.2 The test .......................................................................................................... 34
    8.3.3 Conclusion for removing/exchanging functionality ..................................... 34
  8.4 Evaluation of exchanging a GUI ..................................................................... 35
    8.4.1 Methods ........................................................................................................ 35
    8.4.2 The test .......................................................................................................... 36
    8.4.3 Conclusion for exchanging a GUI ................................................................. 36
  8.5 Evaluation of customer support ..................................................................... 37
    8.5.1 Methods ........................................................................................................ 37
    8.5.2 The test .......................................................................................................... 37
    8.5.3 Conclusion for customer support ................................................................. 37
  8.6 Evaluation of installation aid ........................................................................ 38
    8.6.1 Methods ........................................................................................................ 38
    8.6.2 The test .......................................................................................................... 38
    8.6.3 Conclusion for installation aid ..................................................................... 38
  8.7 Result of evaluation with the test application .............................................. 38
  8.8 Summary .......................................................................................................... 40

9. Results and conclusions ..................................................................................... 42
  9.1 Summary .......................................................................................................... 42
  9.2 Result of the work ............................................................................................ 43
  9.3 Future work ...................................................................................................... 43

10. Sources .............................................................................................................. 44
  10.1 Reference ......................................................................................................... 44
  10.2 Literature ......................................................................................................... 44
  10.3 Internet ............................................................................................................ 44

Appendix A: (Code Examples) .............................................................................. 45
  A.1 TCL/Java .......................................................................................................... 45
    A.1.1 Syntax description ....................................................................................... 45
    A.1.2 Creating a simple GUI from Tcl/Java ......................................................... 45
    A.1.3 Scripting a program modification ............................................................... 45
    A.1.4 Execute a script from Java code ................................................................. 46
    A.1.5 Creation of a new TCL command .............................................................. 47
    A.1.6 Variable passing between scripts .............................................................. 47
  A.2 JavaScript (Rhino) ......................................................................................... 48
    A.2.1 Syntax description ....................................................................................... 48
    A.2.2 Creating a simple GUI .............................................................................. 48
    A.2.3 Scripting a program modification ............................................................... 48
1. Introduction

1.1 Background and purpose
Tietoenator Public & Healthcare AB has a close cooperation with its clients and offer solutions that make their daily work more efficient. They have developed different Java applications that solve all these customers’ needs. These applications are built with different modules of functionality that together build the whole application, much like a method called “normal configuration management method”.

All these modules can be reused to fit the new needs even if some modification is required. The only problem with this method is that it can crave great knowledge about the different modules to be able to reuse or modify them. This will therefore become expensive to situations like adding, removing, exchanging or integrating functionality even if all functionality exists. The report will therefore explore the usage of different scripting methods for minimizing these costs, and compare it to both “normal configuration management method” and “enterprise application integration”.

·
2. Defining the work

This report will first explore other methods than scripting for minimizing the cost of modifying pre built applications to new needs. All these methods including the found scripting methods will be compared to each other, to see which one that works best. A comparison will be made in criteria that achieves such modifications and in other areas that are important to Tietoenator Public & Healthcare AB.

These criteria are:
- Add or remove functionality from an application
- Integrating pre built applications
- Exchange functionality or a GUI
- Customer support and installation aid

The scripting methods will be tested with the help of a pre built application that Tietoenator has. All tests will discover problems and weaknesses with the scripting solutions in these defined criteria. The final decision will take all these things in consideration and compare this to other methods before the conclusion is made, i.e. if scripting is a good solution to the defined problems.

2.1 Scripting tools and report limitations

The report will explore more than one scripting method within the defined criteria since one could have drawbacks compared to others. Tietoenator has their majority of applications built in Java since the system independency is an important factor. This will confine this report to only evaluate solutions that also are system independent and can handle Java objects, i.e. pre built Java applications. Windows only solutions are therefore not good enough to evaluate.

After some investigations where the below scripting methods found:
- Tcl
- JavaScript (An JavaScript implementation called Rhino)
- Visual Basic scripting via COM objects
- Php
- C#

Tcl and JavaScript have no problem with system independency and are therefore chosen for future investigation. Visual Basic scripting and C# has no universal solution that works over various platforms and is therefore excluded. PHP are also excluded since the found extension module that can handle Java objects where in an early beta stage.

2.2 Other methods that scripting are evaluated against

This report has taken two other methods in consideration for achieving the issues defined in 2.1. The methods are Normal Configuration Management Method (NCMM) and Enterprise Application Integration (EAI). These methods use different techniques to solve the criteria and this is better described in chapter 4 and 5.
2.3 Test program

A pre built Java program is used to exemplify how the different scripting methods can be used. The program handles distributed documents in a multi user environment. It uses a web server to provide the user with an interface, shown below in figure 2.3-1. This interface can search, create, delete and edit distributed files.

![Figure 2.3-1: The test-program's GUI.](image)

This program will be used to evaluate the different scripting methods. The evaluation will use a new GUI and evaluate how well scripting can be used with this.

The GUI will present the content of the program with the help of different script-files. It executes script-files, which returns data that shall be present for the user in some way. The test program will not be modified and can therefore still be used normally after this work is done.

![Figure 2.3-2: Description of how the evaluation of scripting with the test program will be done.](image)
3. Definitions and concepts

This chapter will define definitions and concepts that this report uses.

**Java** is a language for writing application like C or C++. Reference literature that has been used in this report is found at [http://java.sun.com/]

**Java interpreter** is an executable that interprets compiled source-code files, i.e. it executes a built application.

**Class-file** is a compiled source-code file that a compiler has built. A built application has at least one compiled class-file per source-code class.

**Jode** is an application that both contain a decompiler and an optimizer for java. The decompiler can transform compiled class-files back into source-code and the optimizer can make compiled class-files hard to understand if decompiled. Reference literature that has been used in this report is found at [http://jode.sourceforge.net/]

**JexePack** is an application that can transform a Java application into an executable for Windows. Reference literature that has been used in this report is found at [http://www.duckware.com/jexepack/]

**J-Integra** is an application that solves the usage of Java objects in Visual Basic code (a script language for Windows). Reference literature that has been used in this report is found at [http://j-integra.intrinsyc.com/]

**JavaScript** is a scripting language that are normally used for enhancing web pages. Reference literature that has been used in this report is found at [Jesper Ek & Ulrika Eriksson - JavaScript 1.3-programmering]

**Rhino** is an implementation of JavaScript that can handle and manipulate Java objects. Reference literature that has been used in this report is found at [http://www.mozilla.org/rhino/]

**Tcl** (Tool Command Language) is a scripting language that is portable through many platforms like MacOS X, Windows, PDA Systems and POSIX. Reference literature that has been used in this report is found at [http://www.tcl.tk]

**Tcl/Java** is an extension to Tcl that are built for handling Java objects and their functionality. Reference literature that has been used in this report is found at [http://tcljava.sourceforge.net/]

**VisualBasic** is a scripting language that is popular among developers just like Tcl and JavaScript. Reference literature that has been used in this report is found at [Paul R. Reed, Jr. - Developing Applications with Visual Basic and UML]

**HTML** (Hyper Text Markup Language) is normally used for web development.
**Php** is a scripting language that is especially suited for web development and can easily be embedded with html.

**Php-java-bridge** is a php module that lets script communicate with a running java "application-server" or a dedicated java process. Reference literature that has been used in this report is found at [http://php-java-bridge.sourceforge.net/]

**Execution path** is the applications functionality path, i.e. the source-codes commands that are executed in an orderly way.

**GUI (Graphical User Interface)** is the interface that a user sees when using application functionality.

**NCMM (Normal Configuration Management Method)** is a development method for assurance than the created objects can be reused to other solutions and minimizing the applications development cost. Reference literature that has been used in this report is found in [Edware H. Bersoff, Vilas D. Henderson, Stanley G. Siegel - *Software Configuration Management, An Investment in Product Integrity*]  

**SCI (Software Configuration Identification)** is a module of grouped functionality that the normal configuration management method defines.

**EAI (Enterprise Application Integration)** is a method for integrating different software applications, i.e. making them flexible and fit different needs. Reference literature that has been used in this report is found in [David S. Linthicum - *Enterprise Application Integration*] and [David S. Linthicum - *B2B Application Integration*]
4. Normal Configuration Management Method

NCMM is a method that defines how an application should be built. It defines criteria for assuring that the application or some part of it can be used to other situations and tries to make the implementation process effective.

4.1 Application life cycle

An applications life cycle is a process that starts with an idea and never finishes until the application is thrown away. The cycle starts with an analyse stage where all demands are formally established and solved with an undefined method.

“The system life cycle begins when a system is conceived to satisfy a need. At this stage of the life cycle, the system may be viewed as a “black box” or large transfer function. How it will work is as yet unimportant. It simply takes a single input (the need or problem) and produces a single output (the solution).”
(Bersoff, Henderson & Siegel, 1980, p. 41)

The next phase is the design stage. It specifies in which module that functionality should exist. These modules can be reused for other purposes if the application design is good and minimize future development costs.

“Once the list of things the system should do is firmly established, how the system should do it is then addressed. At this point in its life, the system begins to become differentiable into its components parts, such as hardware, software, procedures, and so on.”
(Bersoff, Henderson & Siegel, 1980, p. 41)

After the design has grouped all functionality into application modules can the implementation stage start. This means to carry out the instruction formed at the previous stages and the result is a complete application.

“Once the detailed design stage is completed, and not until the design is approved, software program development should begin. At this point we enter the first article development stage of the life cycle.”
(Bersoff, Henderson & Siegel, 1980, p. 42)

The last stage of the applications life cycle begins after the application is built and installed on a client machine. It is called the operational state of the application and focuses on solving encountered problems.

“We now enter the production/deployment stage of the system life cycle, during which “carbon copies” of the updated first article are made and delivered to the customer. Once installed and accepted, the system enters its operational state and our interests in the system life cycle shifts emphasis to maintenance activities.”
(Bersoff, Henderson & Siegel, 1980, p. 43)
A system or application will always encounter modification requests. For not destroying the application while modification is in progress must certain rules be enforced. These rules make this process time-consuming and expensive but are necessary if the product should keep its integrity.

“No matter where you are in a system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.”
(Bersoff, Henderson & Siegel, 1980, p. 43)

“Once the system is built, or even while it is being built, the impact of change is even more dramatic. Not only must the change be validated, simulated and documented, but the production process may have to be interrupted and software modified to accomplish the required modification.”
(Bersoff, Henderson & Siegel, 1980, p. 46)

### 4.2 Software Configuration Identifications

A software configuration identification (SCI) object is a module of grouped functionality, like a class module that holds predefined functionality. During development is the SCIs used for establishing a baseline of completed items and makes the work more efficient. These objects can be used to other situations after the development is finished, described in chapter 4.3.

“The basic purpose of configuration identification is to facilitate the partitioning of baselines into progressively large numbers of SCIs as the life cycle is traversed. The basic motivation for this progressive partitioning is that, frequently, some portions of the software may stabilize before others. To economize and focus management efforts, it is useful to establish a means for distinguishing the stabilized portions from the other portions. If, for example, management knows that pieces X, Y, and Z of baseline update have not changed since the previous update, then management can focus its attention on the pieces other than X, Y, and Z that have changed.”
(Bersoff, Henderson & Siegel, 1980, p. 108)

The built SCI objects that were created during the development stage of the application are then combined to a whole application, shown in figure 4.2-1.

![Figure 4.2-1: Description of how all built SCI objects build the whole application.](image-url)
4.3 The usage of SCI to solve the defined test criteria

The system/application is always challenged with further expansion attempts, modification or to be re-used in other situations after it has been developed, as stated in 4.1. These things require a lot of work for not destroying any application in the process. The application has certain demands on its SCI objects. This can easily destroy an application if any of those demands are changed. A strict version control must therefore be enforced on these SCI objects and every application must record which version it uses. If these rules are upheld can this method solve the defined criteria in chapter 2.

4.3.1 Add or remove functionality from an application

Adding or removing functionality with the normal configuration management method is handled with rebuilding the desired SCI objects. The method handles important things like version control on the SCI objects and prohibits ugly hack solutions. This method has a problem with applications that are already installed on a client machine. The method require that the application is rebuilt and reinstalled for the modification to work. This can lead to a great cost for not loosing any data that was associated to the original application. The installation process must therefore be able to import all created data that the original application had. Adding or removing functionality has therefore an additional cost, along with the cost of making the modification, that can be huge in some cases.

4.3.2 Integrating pre built applications

Normal configuration management method handles application integration with combining the applications already built SCI objects. This method is the simplest way to achieve the desired affect with a minimum development costs. The method has only one problem. It demands a great knowledge of the different applications so that nothing is destroyed in the process. If the SCI objects are modified with the sole purpose of integration in mind are they not reusable for other situations. They are grouped together with the focus of application integration instead of what functionality they hold. A better way to handle this is to start a new development process and then reuse whatever existing SCIs that fit the new design. This
is more expensive since it demands a completely new development cycle but gives reusable SCI objects.

4.3.3 Exchange functionality or a GUI

The normal configuration management method handles the exchange of functionality nicely. All things that should be exchanged are normally grouped together in one or two SCI objects. The affected SCI objects are then exchanged with another modified version. A strict version control is really important here since the application could otherwise lose its intended functionality. The GUI can be a little harder to exchange since it is normally more integrated in the application. This is therefore more or less a static exchange and can not be reused for the next one. Both exchanges require a great investigation cost just to see the affects it brings and to identify what to remove. This process can therefore become expensive in the long run.

4.3.4 Customer support and installation aid

The normal configuration management method has no easy solution to these problems since it normally focuses on the building stage of the application. Customer support occurs when an application is installed on a client machine and installation aid is confined to the installation stage. When an application is running on different platforms can different problem occur. Different systems have difference in system specific issues and can create a problem to customer support and installation aid. During the installation stage can problems be solved by manually redo the encountered problem in another way, or by rebuild the installation application every time it happens. Customer support almost certainly requires a rebuild of the application after the found problem has a solution. Both solutions are very time-consuming and are therefore the problem with NCMM.
5. Middleware and Enterprise Application Integration (EAI)

Middleware technology and EAI can be used to integrate and modify applications, just like NCMM. Middleware uses some logic between the applications to combine their data and EAI focuses on both data and the business processes. This chapter will describe how they can be used to solve the issues defined in chapter 2.

5.1 Middleware

Middleware can be used to connect different applications and servers without the need of knowing what operating system the applications run on.

“In the world of distributed computing, middleware provides a means to connect clients to servers, clients to clients, and servers to servers without the need to navigate through many operating systems, networks, or resource server layers.”

(David S. Linthicum, 2001, p. 18)

Traditional middleware technologies focus on combining different data sources and normally use a message-passing technology to achieve this.

"Traditional middleware technology, such as message-queuing software, ties applications together, but these "point-to-point" solutions create single links between many applications, as we mentioned previously. As a result, the integration solution itself may become more expensive to maintain than the applications it's connecting."

(David S. Linthicum, 2000, p. 8)

Figure 5.1-1: Description of a normal middleware solution. The applications use a message passing system for querying data. A middleware intercepts the message and transforms it to something else.
This method can be used to both combine different applications functionality and form a virtual system.

"Middleware provides developers with an easy way to integrate external resources using a common set of application services."
(David S. Linthicum, 2000, p. 21)

"Middleware provide developers and application architects with the ability to bridge many different systems, bringing them together to form virtual systems."
(David S. Linthicum, 2000, p. 21)

5.2 EAI

EAI has a slightly different focus than traditional middleware solutions. It focuses on the whole business process and its data. This method can therefore be better used to the defined criteria in chapter 2.

"EAI focuses on the integration of both business-level process and data, whereas the traditional middleware approach is data oriented."
(David S. Linthicum, 2000, p. 5)

There exist three different types of EAI and they are:
- Data-level EAI
- Application interface-level EAI
- Method-level EAI

![Figure 5.2-1: Description of the different types of EAI.](image-url)
5.2.1 Data-level EAI

Data-level EAI combine data from different applications and forms a kind of integration between them. One application fetches data that are modified before it is sent back to the source, much like the middleware solution.

"Data-level EAI is the process-and the techniques and technology-of moving data between data stores. This can be described as extracting information from one database, perhaps processing that information as needed, and updating it in another database."
(David S. Linthicum, 2000, p. 18)

"Accessing data in the context of EAI requires an "end run" around application logic and user interfaces in order to extract or load data directly into the database through an interface (see Figure 2.1)."
(David S. Linthicum, 2000, p. 24)

This method is very cost effective since no modification is done to the original applications. The only problem is that the application developers must fully understand all the data flows through the companies’ enterprise. This can otherwise lead to many connections to different kind of data sources and lead to an impossible solution to expand with new functionality.

"The advantage of data-level EAI is the cost of using this approach. Because we are largely leaving the application alone, and not changing code, we don't need to incur the expense of changing, testing, and deploying the application."
(David S. Linthicum, 2000, p. 18)

"Migrating data from one database to another may sound simple and reasonable, but for data-level EAI to work, architects and developers need to understand the complex world of database technology as well as the way information flows throughout an enterprise."
(David S. Linthicum, 2000, p. 23)

5.2.2 Application interface-level EAI

Application interface-level EAI uses the applications interface for the integration. The logic behind all applications is already built and can therefore be reused to achieve the integration. All functionality that an application holds are normally exposed through different interfaces. The developers have made these interfaces because they could reuse them to other situations. This saves a lot of time since they can then focus on building new features.

"Application interface-level EAI refers to the leveraging of interfaces exposed by customer or packed applications. Developers leverage these interfaces to access both business processes and simple information. Using these interfaces, developers are able to bundle many applications together, allowing them to share business logical information. The only limitations that developers face are the specific features and functions of the application interfaces."
(David S. Linthicum, 2000, p. 18-19)
“Developers expose these interfaces for two reasons. The first is to provide access to the business processes and data encapsulated within their applications without requiring other developers to invoke the user interface or to go directly to the database.”
(David S. Linthicum, 2001, p. 52)

This integration can easily be done with programming languages that exposes its interface and separates it from the process and data. Many solutions have been developed to achieve this and most of them use a message broker, shown in figure 5.2.2-1.

![Figure 5.2.2-1: Description of applications that use a message broker.](image)

"This type of EAI is most applicable to packaged applications such as SAP, PeopleSoft, and Baan, which all expose interfaces into their processes and data, but do so in very different ways. In order to integrate those systems with others in the enterprise, we must use these interfaces to access both processes and data, extract the information, place it in a format understandable by the target application, and transmit the information. While many different types of technologies can do this, message brokers seem to be the preferred solution."
(David S. Linthicum, 2000, p. 19)

Application interface-level EAI can still be used with any application that does not use a message broker. This involves making another application interface. Building a new interface to a pre built application is normally very time consuming, but solves the problem in a nice way.

"Assume that the data-level EAI solution won't work due to the complexity of the databases and the binding of logic to the data. Thus, in order to integrate the old and new applications, the natural point of integration should be application interfaces."
(David S. Linthicum, 2000, p. 40)
"Due to the development and testing costs, building interfaces into existing applications is generally very expensive. Other options are to employ user interface-level EAI, where the user interface (screens) is the point of integration."
(David S. Linthicum, 2000, p. 41)

5.2.3 Method-level EAI
Method-level EAI exposes all business logic like functions and methods so it can easily be reused to new purposes. This saves a lot of time for developers since they do not have to rewrite functionality for every modification/integration.

"Method-level EAI is the sharing of the business logic that may exist within the enterprise. For example, the method for updating a customer record may be accessed from any number of applications, and may access each other's methods without having to rewrite each method the respective application."
(David S. Linthicum, 2000, p. 19)

There exist two different approaches to solve this. The first one uses application servers and the other involves distributed objects.

"The mechanisms to share methods among applications are numerous, including distributed objects, application servers, TP (transaction processing) monitors, frameworks, and simply creating a new application that's the combination of two or more applications. There are two basic approaches: You may create a shared set of application servers that exist on a shared physical server, such as an application server, or you may share methods already existing inside of applications using distributed method-sharing technology such as distributed objects."
(David S. Linthicum, 2000, p. 19-20)
An application must sometimes be rebuilt to support these approaches. This can be a costly procedure and creates strong connections between the involved applications. These connections can create problem for the developers if the application should be reused in future development.

"Unlike other EAI levels, there is little choice here other than to rebuild the applications so that they support method-level EAI. To accomplish this, we have two options with method-level EAI. First, we can move much of the business logic to shared server, such as an application server. Second, we can rebuild each application using method-sharing mechanism, such as distributed object technology, to create a tightly coupled application that allows easy cross-access of methods."

(David S. Linthicum, 2000, p. 63)

5.3 The usage of EAI to solve the defined test criteria

Adding or removing application functionality with EAI can be solved if the application is built in business models. The developer can then rebuild the desired models and enforce a strict versioning control to achieve the effect. Integration of applications can be handled with either data- or method-level EAI. The choice of method depends on what the integration should integrate. Exchanging an application GUI with EAI requires that the application expose its interfaces, i.e. expose its functionality through easily used interfaces. The customer support and installation aid issue cannot be solved with EAI.

If an application uses some sort of EAI can it even make the customer support issue harder to solve.

The EAI method creates in general a complex solution to application integration and modification. This depends on the fact that it creates another element that can fail at runtime. The only time this method do not create problem is when the method-level EAI is used. This uses the created application/business methods and builds a completely new application. Method-level and application-level EAI have only one downside; they both demand that the applications involved are built in a special way. This can therefore demand a great development cost for making them fit this specific requirement, if they are not already built that way.

5.3.1 Add, remove or exchange functionality from an application

These issues are easiest solved with an application that is built in business models. The developer could then just modify the models and achieve the desired effect. To rebuild a pre built application into business models can be a time consuming job since it most certainly requires total modification of the code and application flow.

5.3.2 Integrating pre built applications

The integration with EAI can be handled with data- or method-level integration. Data-level integration is best used if the applications data should integrate, otherwise must the method-level be used. Method-level requires that the applications are built in business models and that it expose its functionality. This normally requires a great deal of work if the applications are
not already built to support this. Therefore is this method not a good solution to pre built applications. Data-level integration leaves the original application alone but makes a strong connection between them. In worst could this prohibit future development and are therefore not a perfect solution to this.

5.3.3 Exchange a GUI
To achieve a GUI exchange with EAI must the application either has its functionality exposed or use a message broker. This is a big modification to the application if it is not built like that.

5.3.4 Customer support and installation aid
These issues cannot be handled with this method. The usage of EAI with an application will even make this issue harder to solve since it introduces a new level that could fail at runtime. This is therefore better solved with normal programming methods like debugging and rebuilding the application.
6. Scripting tools
This chapter will describe the found scripting methods that can be used with pre built Java-applications.

6.1 Tcl
Tcl is a scripting language that offers methods for creating applications, similar to Java and C. It is mainly used for tying together applications, protocols, devices and frameworks. Tcl has also a toolkit that provides a fast and powerful way to create GUI applications that run on Windows, Unix and Macintosh. This report will focus on an extension module called Tcl/Java, since it can handle Java objects.

6.1.1 Description of Tcl/Java
Tcl/Java offers two different ways to access Java functionality with methods that are called “tclBlend” and “jacl”. TclBlend is an extension to Tcl that communicates with Java objects through loading a Java interpreter into an existing tcl process. Jacl is in contrast a self-contained implementation of a Tcl interpreter written entirely in Java. This means that jacl is normally used to incorporate scripting functionality into an existing Java application. TclBlend is mostly used through a shell like environment and this report is based on the 1.3.1 version of Tcl/Java.

Both these methods are free to use, copy, modify and distribute provided that its copyright notice and the two paragraphs that exists in the licence file appears in all copies of the software.

6.1.2 Usage of tclBlend and jacl
The syntax of both tclBlend and jacl are identical with the exception of some internal commands that only tclBlend uses. This description will not cover these commands since they are few and not important to the defined criteria. Both methods have therefore the same syntax and will not be handled differently in this report.
Both Tcl/Java methods can use pure tcl-commands and all functionality that an ordinary Java interpreter offers. This means that it can create Java-objects, use functionality that classes holds and access variables through its own script interpreter. The syntax structure is based on tcl, which to an ordinary Java programmer only differs in some minor ways. Variables that script defines cannot be accessed globally until they are defined as “global”, shown in appendix A.1.6. In tcl are a function defined as a process. Global variables can therefore not be accessed from these functions and must instead be passed as parameters to it. A more complete list of commands can be viewed in appendix A.1.1.

There are some prior concerns that must be dealt with before Tcl/Java can be used. Besides a working installation of Java does Jacl require that “jacl.jar” and “tcljava.jar”, which was created during compilation, is accessible from the Java interpreter. The Java application could then run the made scripts through these jar-files functionality. TclBlend has more requirements that must be fulfilled before it can be used. Since it is based on Tcl and should use Java objects, it needs a working copy of Tcl installed with a thread extension. Tcl can then load the extension that handles Java objects into its own process. This is more complicated than Jacl but have some benefits since tclBlend have for example a more debug-friendly environment.

6.2 JavaScript

JavaScript is normally used to enhance web pages. It cannot easily be used to the criteria defined in chapter 2. A solution has therefore been developed to solve this problem and this method is called Rhino. The method is based on JavaScript and is enhanced with the ability to handle real Java objects.

6.2.1 Description of Rhino

Rhino is an open-source implementation of JavaScript built entirely in Java. It has the ability to evaluate JavaScript functionality and at the same time use ordinary Java functionality. Rhino has also a feature that can compile its script-files into class-files. The implementation offers methods that can be used from Java code and a shell-like environment where script-files can be evaluated. This report is based on version 1.5R1.4 of Rhino.
Rhinos source code is distributed under Netscape Public Licence 1.1 (NPL 1.1) and can therefore be used without any problems.

6.2.2 Usage of Rhino
The syntax of Rhino is very similar to ordinary Java code since it is based on JavaScript. It only differs when accessing own packages. They are accessible through a virtual package called “Packages”. If a script should access classes that are accessible from the interpreter it is done through this package, i.e. “Packages.<className>”. Variables that are created do not have to define a type and can be accessed within its own context, i.e. the scripting environment class. A function is created with the syntax: “function <nameOfFunction>(…)”. A more complete list of basic Rhino syntax can be viewed in appendix A.2.1.

Rhino has an ability to compile its script-files into class-files. These files cannot be handled as regular compiled class-files. They cannot for example be created with the “new” command from Java code. Instead could these files be used through a normal Java interpreter since they contains a main method. Such files are easier to handle compared to the usage of uncompiled script-files. These files are used through functionality that the Rhino jar-file contains. All this is better shown in appendix A.2.5.

The requirement that must be fulfilled before Rhino can be used is an installation of Java later than version 1.1. For the usage of JavaAdapter features or an optimization level greater or equal to zero must it also run under a security manager that allows the definition of class-loaders. All this is fulfilled when the sun version of Java is used, which makes this a system independent solution. When these requirements are met, all needed for this method to work are that its jar-file is accessible from the Java interpreter. This jar-file can be downloaded binary or in source code from the developer site.

6.3 Description of Visual Basic and Java
Visual Basic has the ability to communicate with other components via COM objects. These objects are normally bound to the Windows® operating system but have in some cases also been ported to Linux.

"Component Object Model (COM) is Microsoft's middleware plumbing that allows components to communicate."
(Paul R. Reed, JR, (1999), p. 469)

"COM has been ported to different flavours of the UNIX operation system as well. Unfortunately, enhancements to COM and its distributed flavor, DCOM, are not always released in unison to all platforms. This makes for a difficult integration strategy at best."
(Paul R. Reed, JR, (1999), p. 469)

Visual Basic can therefore communicate with Java-objects through these COM objects. One found solution that solves this is called J-Integra® (version 1.6). This method is not free to use and is therefore not a good solution to these problems. J-Integra is also not usable outside of
Windows®, i.e. it is not a system-independent solution. There exists other Basic solutions for Linux systems but their syntax is not 100% compatible. This report will therefore exclude this method as a solution to the defined criteria in chapter 2.

6.4 Description of Php

There exists a php scripting method that can be used with Java-objects and it is called php-java-bridge. This method are still in an early development stage and cannot be called stable. Therefore are this method excluded from this report.
7. The effects of scripting with an application

When scripting is used with an application, it creates several issues and questions. This chapter will try to define and solve them all.

7.1 Do scripting break the Java object structure

The object structure that a Java-application holds is upheld by protecting vital functions and data by using “private” and “protected” identifiers in the source code. These rules are still upheld when scripting is used since all tested methods use a regular Java-interpreter for accessing Java-functionality. The only thing that script can use is public functions, data and static defined things. This means that the only thing script can break is the executions path of a application, shown in appendix A.1.3.

For example, neither a script nor a Java class can access a private function or data in a class shown below in example 7.3-1.

A Java application

```java
public final class User {
    private String user = “a_userName”;
    private String password = “a_password”;
    ...
    protected User(String u_name) {
        user = u_name;
    ...
    }
    public static User getUser(String u_name, String u_pass) {
        if (verifyUser(u_name, u_pass))
            return new User(u_name);
        else
            return null;
    }
    ...
}

public final class DoThings{
    public DoThings() {
        ...
    }
    public void printSecretStuff(User user) {
        //   User user = new User("root");  /* Error */
        if (user != null) {
            System.out.println("a_secret_string");
        }
    }
    ...
}
```

Rhino-script

```javascript
var user = Packages.User.getUser("root", "root"); #OK
var dt = new Packages.DoThings(); #OK
dt.printSecretStuff(user); #OK
var user2 = new Packages.User("username"); #Error (protected function)
print(user2.passwd); #Error (private class member)
```

Example 7.3-1:
Script example of how it only can access public defined classes and objects
7.2 The effect of unauthorised editing of script-files

Users that have access to an application that uses scripting can easily modify these files. This modification is one of the benefits with the scripting but could also be exploited in various ways. A logging device could for example be added to a script-file that logs usernames and passwords, shown in figure 7.2-1.

![Diagram](link)

Figure 7.2-1:
Description on how an unauthorized logging device could be made with the help of a script interface.

Unauthorised editing could also be exploited to either fool user that functionality called was done or just to crash the application. Both these threats could lead to loss of data and a great restoration cost.

Another possible situation is when users are prone to make own solutions with the help of pre built application. If scripting is used with an application could a user see an opportunity to modify its scripts and solve some other problem. This could lead to application error even if the intention was good and a lot of recovery work to restore application data.

All these threats have to do with the fact that script-files are evaluated in clear text. A solution to these issues is to use Rhino method since it could compile its script into class-files. If such a file should be modified it must first be decompiled, rewritten and then compiled back to the class script-files. It is not likely that an end user can know about this possibility to edit such a file since it looks like a class-file, i.e. like the rest of the application.

7.3 The long term affect with scripting

The two found scripting methods could be used with an application in two ways. They can either use the applications functionality directly from the script or integrate itself into the applications source-code. The first solution could never damage the original application. It only uses desired functionality from the application and does not interfere with its normal execution. When script functionality is used from the application source code, it could prohibit further development of the application. The application source-code evaluates different script-files and makes the execution path hard to follow. When the application depends on external scripts it also makes an assumption on what it does. These assumptions could easily be broken if the files are modified and could in worst-case crash the application. This could for example happen if the application assumes that a script should return some data. If this data was changed could the application in worst-case crash. This problem has no other solution than to not use script with the application.
7.4 Scripting and program security

Script-files are normally evaluated in clear text format and could create a security problem when used with an application. The script-files can give a curious reader a hint on how the application is using the code to achieve its functionality. This information can used for future investigation on how to break program security or extract secret information from it.

It is a well-known fact that a compiled Java-class could be decompiled into source-code with the help of different programs. Other programs have therefore been developed to distorting these files. These programs are called “obfuscation programs” and can among other things exchange function-calls and variable names in chosen class-files. A decompilation of such a file shows that it is nearly impossible for a human to understand, shown in appendix A.4. These programs do not work well with scripting because they only handle regular class-files and not script-files. Since the script-files are left unmodified will they stop to work, i.e. they try to call classes with names that have been changed. This means that the made scripts must be modified manually to use the exchanged names. Modify this by hand is not a solution to the problem since it is nearly impossible to do. If the obfuscator instead is used to distort all class-files but the application class interface, we get an almost secure application that also could be used by the scripts. Then if someone tries to decompile such a class-file from that application, the interface class function-calls are the only readable code he or she could see. This means that we can protect interface functionality and only leave its method name visible to someone that tries to decompile such a file.

As said before, Rhino could compile its script-files into class-files. These class-files cannot be added to the obfuscator because they are not regular class-files, shown in appendix A.2.5. This is therefore also not a solution to this problem.

Another way to protect a Java program is to make an executable with the compiled classes and the help of for example “JexePack”. This application makes an executable from all classes that a Java application holds. It completely protects the application to all decompilations. This method has a major drawback since the application loses the system independency at the same time. The executable is only able to run under Windows® and disables all chances for scripting to be used with the application.
8. Evaluation of the found scripting tools

This chapter will describe the evaluation of the defined criteria in chapter 2 with the test application. It will also grade all the methods in these criteria and decide if scripting is a good solution compared to NCMM and EAI.

8.1 Evaluation of integrating applications

8.1.1 Methods

This issue can be divided into two methods. The first one involves creating a script interface to each application. Each such interface has different script-files where each file holds one functionality to its application. These can then be combined in various ways to achieve a completely integrated application. A master script can then easily execute these script-files and achieve the integration, shown in figure 8.1.1-1. The old application can still run at the same time as the integrated one since the scripts only expand the applications with another execution path. There exist some situations when this method cannot be used, like when the scripting cannot access the execution starting point of the application. This can occur when the application requires a web-server to execute, i.e. when the application is a servlet and runs through a web-server. The solution to this is to let one script interface be integrated into one applications source-code, shown in figure 8.1.1-2.

![Diagram of a script integration. All applications still work individually.](image1)

![Diagram of a script integration where application1:s source code has been modified to handle scripts. Application 2 will continue to work normally.](image2)
An application that already has a script interface could easily be reused to achieve the integration. It can easily be modified to fit a new integration since the scripts are normally used in clear text format. The solution in figure 8.1.1-2 gives more application modularity than the one in figure 8.1.1-1. This mainly depends on the fact that an application that uses external script functionality almost certainly wants the data it produces. The application could easily receive other data than expected if the script-files are changed. This could in worst-case lead to an application crash and is therefore more of a static solution.

Data must be passed between the script-files when they only contain one functionality to the application each. The only way to accomplish this with the found scripting methods is by letting a variable hold it. Another method for passing data exists if the application uses external script-files. The script-files could return it back to the application instead of leaving it in a variable. This solution is preferable since leaving data in a variable also requires it to be global. The usage of script-files is simple when all this is set up. This means that only a little effort will be required to understand what these script interface internally do, and the programmer could instead focus on the usage of different script-files to achieve the required functionality.

8.1.2 Script evaluation with the test application
A Java file-browser was written to test this issue in Java code. This application lists the content of the test application in a new way. The simplest way was to let the file-browser evaluate external script-files, described in figure 8.1.2-1. This depends on the fact that the file-browser was not already built and could therefore integrate the script usage into its source-code without any conflicts.

8.1.3 Conclusion for integrating applications
This issue was implemented in both Rhino and Tcl/Java. Both methods could achieve the desired task surprisingly easy with very little work. They will therefore start this test with full points.
Evaluation of how the scripting affect the application must then be done. The long-term effect of scripting affect both methods just as much, as described in chapter 7.3. Both methods will therefore lose one point to this fact. Rhino can handle the security issues better than Tcl/Java, as described in chapter 7.4. Tcl/Java will therefore lose one point in this test since it has no method for handling this problem.

Chapter 4 described a integration with the normal configuration management method and it said that the only problem is the high development costs. This method will therefore get four points here.

The EAI method in chapter 5 described how it could handle the application integration. The method prohibits either future development or lead to a great deal of work depending on what solution that was chosen. On the other hand can this method handle the integration with new application with minimal work if it is built the right way. This method will therefore get four points here.

<table>
<thead>
<tr>
<th>Integrating applications</th>
<th>Rhino</th>
<th>Tcl/Java</th>
<th>NCMM</th>
<th>EAI</th>
</tr>
</thead>
</table>

Table 8.1.3-1:
Summarizing table of the integration (1=didn’t work, 5=no encountered problems)

### 8.2 Evaluation of adding functionality

#### 8.2.1 Methods
Scripting can solve this by letting the scripts execute the applications functionality, shown in figure 8.2.1-1. The original application is kept unmodified with this method. This means that the original application could execute normally at the same time as the script offer a slightly different version of it. Script functionality could also be used from Java source-code, shown in figure 8.2.1-2. The original application must this way modify its source-code so that the original functionality is lost. When the application has modified its source-code is the functionality easily exchangeable by just editing these script-files.
When we talk about adding more functionality to a Java-application with the help of scripting there exist a natural conflict compared to modify its source-code. The conflict lies in the simplicity of the program flow vs. flexibility. Scripting offers the flexibility but complicates the execution path of the application. Therefore are the scripting methods best used here when the solution are not likely to be permanent and do not therefore prohibit future development. The only situation where scripting makes a permanent solution is when it offer the application another execution path, shown in figure 8.2.1-1. This method has not solved all future development issues but they are all confined to the new execution path. The scripting interface does not affect the normal execution path since they are isolated from each other.

### 8.2.2 The test

This issue was tested with the test application and enhanced its functionality with a better file deletion. The functionality was inserted into the file-browser and enhanced it with a structured deletion of files and directories. The original application threw all files in a trashcan without information on where it was originally stored. This problem was solved with one script that marked the selected file or directory as deleted and another to list all deleted files. The made change did not affect the original application since this test used the method that was described in figure 8.2.1-1, i.e. only the file-browser uses this enhanced functionality. Files that the file-browser deletes can also be viewed from the test application since they both use the same subsystem.

### 8.2.3 Conclusion for adding functionality

The test was written in both Rhino and Tcl/Java. Both methods could achieve the desired task surprisingly easy with very little work. They will therefore start this test with full points.

Rhino and Tcl/Java works equally well in this issue. They both have problems with the long-term affects (shown in chapter 7.3) and will therefore lose one points to this. Security is also a problem with both methods since it could give a hacker an entry point to the application, described in chapter 7.4. Rhino’s ability to compile its script-files into class-files is not important here since this solution only should be used for temporary situations. They will therefore both lose another point each here.
Chapter 4 described that NCMM has no drawbacks other than the hard work required to accomplish this. This method will therefore get four points.

The EAI method in chapter 5 described a method for adding functionality that was called method-level EAI. This method has no problem with future development but is hard to achieve if the business models are not already created. On the other hand will this process be smooth if the models already were done and therefore will this method get four points here.

<table>
<thead>
<tr>
<th>Adding functionality</th>
<th>Rhino</th>
<th>Tcl/Java</th>
<th>NCMM</th>
<th>EAI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2.3-1:
Summarizing table of adding functionality (1=didn’t work, 5=no encountered problems)

8.3 Evaluation of removing or exchanging functionality

8.3.1 Methods
Removing or exchanging application functionality with scripting is done with Java’s normal methods. The tested methods have no own methods for accomplishing this.

An exchange of application functionality is therefore done in two steps. The original functionality must first be removed before the new is added. Java has functionality for removing event functionality and can also overload function calls. The removing of event functionality is shown in appendix A.1.3 and A.2.3. Overloading functionality require that functionality in the class is accessible from the scripts, i.e. public. Scripting can solve this nicely since it can offer the application another execution path. This way can the change be done without destroying the applications normal functionality.

8.3.2 The test
This method was not evaluated with the test application. Instead was another small application made to exemplify this solution. The test can be viewed in appendix A.1.3 and A.2.3.

8.3.3 Conclusion for removing/exchanging functionality
Both methods could achieve adding and exchanging functionality surprisingly easy with very little work. They will therefore start this test with full points.

These scripting methods will give the application a great deal of problems down the road. The long-term effects (described in chapter 7.3) create among other things development problems since the execution path could become nearly impossible to follow. Functionality exists in different places and one never knows which is used. These methods will therefore lose three points each because of this. No other great advantage or disadvantage exists for the scripting methods.
Chapter 4 described that the usage of NCMM have no other disadvantage than the hard work that must be done. This method will therefore only get four points here.

The EAI method in chapter 5 described a method for removing or exchanging functionality from an application that used the same method as the adding of functionality. Since EAI solves this like the adding of functionality does will it also receive four points here.

<table>
<thead>
<tr>
<th>Remove/exchange functionality</th>
<th>Rhino</th>
<th>Tcl/Java</th>
<th>NCMM</th>
<th>EAI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3.3-1:
Summarizing table of the removing/exchanging functionality (1=didn’t work, 5=no encountered problems)

8.4 Evaluation of exchanging a GUI

8.4.1 Methods

If scripting should exchange a GUI must the applications functionality first be easily accessible. This can be achieved with a scripting interface and are described in figure 8.4.1-1. Such an interface can either be newly built or reused if it already exists. An existing script-interface can easily be used without a great knowledge of the whole application. This depends on the fact that script is normally used in clear text format and shows how the applications functionality can be accessed.

This method is just like in chapter 8.1.1 best solved with the method where each script-file only holds one functionality per file. They could then be combined in various ways to fit another user interface or to solve other problems, shown in figure 8.4.1-1.

If TCL is used for this exchange could its TK-extension build the whole GUI. This could have some advantage to using normal Java functionality since it does not mess up the usage of different languages as much as Rhino does. All the same, both methods require a build of the new GUI and will therefore require approximately just as much work.
8.4.2 The test

A new GUI was created to evaluate this issue with the test application. This GUI has no fancy structure but is only created for this evaluation. It is a shell like GUI that uses the same script-files as the file-browser.

![Diagram of GUI and script files](image.png)

Figure 8.4.21: Description of the new GUI and how it uses the same script-files as the file browser

8.4.3 Conclusion for exchanging a GUI

This issue was tested in both Rhino and Tcl/Java and they both passed the test without any problem. They will therefore start with full points.

The security issue, described in chapter 7.4, is something that must be taken in consideration here. Rhino has a clear advantage here since this can be used for permanent solutions. It could compile its script-files into hard to modulate class files and hide the fact that scripting is used with the application. Tcl/Java will lose one point in this issue since it has no solution to this. Rhino will loose one point to TCL because TCL can use TK to build the whole GUI. No other thing will affect the usage of these methods.

Chapter 4 described that the NCMM is an expensive solution for this problem. Every exchange of GUI will lead to both a great investigation and development costs. All other things are handled nicely but when the costs are compared to the gain will this method only get two points here.

The EAI method in chapter 5 described a method for exchanging a GUI. This method used either a message broker or distributed objects/exposed functionality to achieve the exchange. Both methods can be easily used when the application is already built to support this. The application must otherwise first be totally rebuilt, which is a costly procedure. This method will therefore only get three points here.

<table>
<thead>
<tr>
<th>Exchanging a GUI</th>
<th>Rhino</th>
<th>Tcl/Java</th>
<th>NCMM</th>
<th>EAI</th>
</tr>
</thead>
</table>

Table 8.4.3-1: Summarizing table of exchanging a GUI (1=didn’t work, 5=no encountered problems)
8.5 Evaluation of customer support

8.5.1 Methods
This issue handles situations that are hard to predict for an application. One solution to this problem is to make an interface that could evaluate newly made script-files. This interface is easy to build since it only needs a few lines of code, shown in appendix A.1.4. Script can be written after the problem is found and then evaluated through this interface as a solution. This also means that scripts could be constructed for an investigation to where a problem exists. If a customer explains what kind of problem they have, the support helper could build a script for investigation and let the customer execute it through this interface.

8.5.2 The test
This issue was implemented in both Rhino and Tcl/Java. It was solved with giving the user a possibility to enter a filename that holds scripting functionality. The file is then evaluated and the output returned to the user as a text result, shown in figure 8.5.2-1.

8.5.3 Conclusion for customer support
This issue is easy to implement in both Rhino and Tcl/Java. Neither of the two have some feature that could be thought as an advantage. For example, the security issue have no major downside since the made scripts are only used for a short time. The long-term effects are also not a problem. In other words, this is the best applications for the scripting methods.

Using NCMM to accomplish this is hard since the application must be able to import newly created classes. It must also be able to execute new functionality from the class-files and will therefore only get two points here.

The EAI method in chapter 5 described that there was no real way for handling this issue. This method will therefore only get one point here.
8.6 Evaluation of installation aid

8.6.1 Methods
A flexible solution is required to this problem since it must handle issues from the installation stage of an application that is hard to predict. This method should be able to use functionality from the installation application in various ways to accomplish a solution. The found solution could later be included in the installation application to handle just such a case.

8.6.2 The test
This issue is a combination of adding and integrating functionality. It requires using existing functionality together with newly created and then integrate them into a single method. These issues are already been dealt with in chapter 8.1 and 8.2 and are therefore not evaluated here again.

8.6.3 Conclusion for installation aid
This issue is easily implemented in both Rhino and TCL. Since they both are used for a short period will the real application not be affected. They will also not affect the applications security and are therefore a very good example of when scripting can be used without further conflicts. Both Rhino and Tcl/Java will receive full points since they both have no disadvantages here.

Using NCMM as a solution to this issue requires that the installation application must be rebuilt every time a problem occurs, i.e. the development cycle must restart from scratch every time a problem is encountered (stated in chapter 4). This method will therefore only get three points to this issue since rewriting the application requires a lot of knowledge and a great amount of work.

The EAI method in chapter 5 described that there was no real way for handling this issue. This method will therefore only get one point.

8.7 Result of evaluation with the test application
No evaluations with the test application affected its original functionality. It instead gave the application a new flexible way to access its functionality. This is done with both the file-browser and the shell like GUI, shown in figure 8.7-1 and 8.7-2. These applications can also
evaluate newly created script functionality through a simple interface. The simplicity of such interface can be viewed in appendix A.1.4 and A.2.4.

Figure 8.7-1: Result of the evaluation. The test application can now also be accessed through a scripting interface.
8.8 Summary

Using the found scripting methods to solve the defined problems in chapter 2 is easily done. Scripting can easily modulate an applications execution path and functionality. The scripts can be modified with easy means since they are normally evaluated in clear text. Situations like testing a solution, making a new interface to an application and making an installation application could therefore be handled without any major conflicts. This means that one could for example get fast feedback to an application modification without a great development cost. In this situations are scripting only used for a short time and do not really affect the original application. Other issues like integration between applications can be solved with both scripting methods but could create problems. It could prohibit future development if one application integrates the script usage to its source code. Both scripting methods can easily handle small application modifications. Rhino is even better than Tcl/Java here and mainly depends on its ability to compile its script-files into class-files. The usage of scripting is then kept secret, which means that most security issues are solved.
NCMM and EAI handle some issues better than the scripting methods. The choice of what methods to use must therefore depend on what issues that is important for the application, now and in the future.

<table>
<thead>
<tr>
<th></th>
<th>Rhino</th>
<th>Tcl/Java</th>
<th>NCMM</th>
<th>EAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrating applications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove/exchange functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exchanging a GUI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer support</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Installation aid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td>23/30</td>
<td>22/30</td>
<td>19/30</td>
<td>17/30</td>
</tr>
</tbody>
</table>

Table 8.8-1: Summarizing table of the problems defined in chapter 2 (1=didn’t work, 5=no encountered problems)
9. Results and conclusions

9.1 Summary

Scripting can offer an inexpensive and flexible way to handle small customer modifications and as an application problem solver. This report has evaluated two scripting method that can be used to this purpose, “TCL/Java” and “Rhino”. These methods can be used in two different ways with an existing application. They can either give an application another execution path or integrate the script usage into the source-code.

Scripting can easily make small application modifications without conflicts to the original application when it makes another execution path. Permanent modifications are the only situation that can create problems for this method. It can easily create an execution path that is hard to follow and prohibit future development of the application.

An applications source-code must use external script functionality with caution since it makes assumptions on what it does and should return. These assumptions cannot be broken since it otherwise will crash the application. This means that the made script-files cannot be modified without these requirements in mind and becomes more or less a static solution. Therefore is this method best usable when the scripting method cannot reach the programs starting point of execution.

Scripting creates security issues when it is used with a program. Since script functionality is evaluated in clear text, it could give a curious reader a hint on how the application security is handled. It could also show how to extract secret data or where to put a logging device for capturing usernames and passwords. The Rhino scripting method could solve some of these issues since it can compile its script-files into class-files. For this method to be exploited must the script-files first be decompiled, edited and compiled back into class-files. It is not likely that a user can figure out that the program uses script-files if it uses this method. Therefore can this method be safely used this way to vulnerable situations.

The consequences of scripting must always be taken in consideration. Issues like customer support and installation aid are therefore a good situation when scripting works best. Other situations like adding or exchanging application functionality can be used for non-permanent situations, i.e. situations that are not likely to be used for future development. It can otherwise create an execution path that is nearly impossible to follow. Integration between applications is, with the same reason, not a good solution to the problem if it is handled wrongly. The only method that can handle this integration without any major problem is the one that that gives the application another execution path. This depends on the fact that it does not become a vital part of the application and can easily be excluded for future development.

Both NCMM and EAI are shown to be better than the scripting methods in certain issues. This shows that the scripting methods can be enhanced further in certain criteria. The scripting methods have a clear advantage to non permanent issues, like customer support and installation aid.

The conclusion of this report is that the scripting methods could easily integrate and modulate applications functionality. It could therefore be easily used to get fast feedback to application
modifications. Sometimes will scripting bring unwanted effects and must therefore be used with caution. The scripting method is therefore a powerful tool to pre built applications when it is used appropriately.

9.2 Result of the work
The result of the work is an application that became enhanced with the flexibility from scripting. It received two new GUI and an easy way to integrate or add new functionality. The work showed how scripting can be used to solve the different problems defined in chapter 2. It also showed weaknesses and how they could be solved.

9.3 Future work
The time confined this report only to evaluate two different scripting methods. Other scripting solutions could be used as a solution to the defined problems in chapter 2. It could be interesting to evaluate for example the php scripting method since it is easy to use with different kind of databases and webpages.

Both tested scripting methods had some issues that could be enhanced. They both have no real way to pass data between external scripts other that letting a variable hold it. This could lead to problem if the variable is not set up properly before a script is called. A better way would be to call the script with the data as input parameter. This is better since there is less a chance that the evaluation of the scripts will fail on faulty data.
10. Sources

10.1 Reference


10.2 Literature


10.3 Internet

http://java.sun.com/
Java, Sun Microsystems, Inc. (August 7, 2003)

http://jode.sourceforge.net/
Jode, Jochen Hoenicke. Canonic (May 29, 2002)

http://www.duckware.com/jexepack/
JexePack, Duckware (November 17, 2003)

http://j-integra.intrinsyc.com/
J-Integra, Intrinsyc (December 1, 2003)

http://www.mozilla.org/rhino/
Rhino, The Mozilla Organization (October 2, 2000)

http://www.tcl.tk/
Tcl, Tcl Developer Xchange (Januari 8, 2004)

http://tcljava.sourceforge.net/
Tcl/Java, Mo DeJong (Cygnus / Red Hat), Scott Redman (Scriptics), Christian Krone (SQL Datenbanksysteme GmbH) (August 7, 2003)
Appendix A: (Code Examples)

A.1 TCL/Java

A.1.1 Syntax description
Below is a translate table between some basic Java commands and TCL/Java.

<table>
<thead>
<tr>
<th>Java Command</th>
<th>TCL/Java Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>String testString = new String(...)</td>
<td>set testString [java::new String &quot;...&quot;]</td>
</tr>
<tr>
<td>int anInt = 8</td>
<td>set anInt 8</td>
</tr>
<tr>
<td>testString.compareTo(...)</td>
<td>$testString compareTo &quot;...&quot;</td>
</tr>
<tr>
<td>while (...) &amp;&amp; ...</td>
<td></td>
</tr>
<tr>
<td>if (...)</td>
<td>if {...}</td>
</tr>
<tr>
<td>if (testString == null)</td>
<td>if {[java::isnull $testString]}</td>
</tr>
<tr>
<td>functionName(..., ...)</td>
<td>proc functionName {... ...}</td>
</tr>
<tr>
<td>&quot;global variables&quot;</td>
<td>global variableName</td>
</tr>
<tr>
<td>return ...</td>
<td>return ...</td>
</tr>
</tbody>
</table>

For full list of usable commands read the TCL/Java manual page at: http://tcljava.sourceforge.net/docs/TclJava/contents.html

A.1.2 Creating a simple GUI from Tcl/Java
This example shows how a graphical user interface could be created from Tcl/Java script.

```java
package require java #required for accessing Java classes
java::import java.awt.GridLayout
java::import -package javax.swing JFrame JButton
set frame [java::new JFrame] #create a frame
set button [java::new JButton "Press me"] #create a button
set contentPane [frame getContentPane]
$contentPane setLayout [java::new GridLayout 3 2]
$contentPane add $button
java::bind $button actionPerformed { puts "hello" }
$frame show #shows the frame
vwait zzz #don’t exit right away
```

A.1.3 Scripting a program modification
This example shows how scripting with Tcl/Java could change a program functionality by giving it another execution path. I.e. it shows how a class-event is substituted to a new functionality. The event comes from the EventClass class and is generated from a button called jButton1.

```java
EventClass.java
public class EventClass extends JFrame {
    ...
    public JButton jButton1 = new JButton("Press me");
    public ActionListener button_event = new ActionListener() { ... };
    //add the button to the frame and use the "button_event" when pressed ...
    ...
```
public static void main(String[] args) {
    ...
}

The script
package require java #required for accessing Java classes
set myEventClass [java::new EventClass] #get the class that holds the event
set theButton [java::field $myEventClass jButton1] #get button that the class holds
set theButtonsActionListener [java::field $myEventClass button_event]
$theButton removeActionListener $theButtonsActionListener #remove the first listener
java::bind $theButton actionPerformed { ... } #bind new action to the button
$myEventClass show #show the frame
vwait zzz #don’t exit until done

This could also be handled by removing the EventClass button and exchanging it with a newly created button from the script.

package require java #required for accessing Java classes
java::import javax.swing.JButton
set myEventClass [java::new EventClass]
set class_button [java::field $myEventClass jButton1]
set script_button [java::new JButton "Press me"] #create a button
set contentPane [$myEventClass getContentPane]
$contentPane remove $class_button
$contentPane add $script_button
java::bind $script_button actionPerformed { … } #shows the frame
vwait zzz #don’t exit right away

A.1.4 Execute a script from Java code
This example shows how scripts could be executed by Java code. The arguments are accessed by the names args0, args1 … argsN from the script.

public class ScriptExecutorInterface {
    ...
    public void execScript(String scriptFile, String[] args) {
        Interp interp = new Interp();
        TclObject scriptArg;
        try {
            for(int i=0; i < args.length(); i++) {
                //set the script args
                scriptArg = tcl.lang.TclString.newInstance(args[i]);
                interp.setVar(new String("arg" + i), scriptArg, TCL.GLOBAL_ONLY);
            }
            String execString = "source " + scriptFile;  //add the "source" to execute a script-file
            interp.eval(execString);
            String returnValue = interp.getResult().toString();
        }
        ...
    }
}
A.1.5 Creation of a new TCL command

The example shows how one can build own commands that could be used through tcl syntax. For this example to work the CLASSPATH variable must be set to the path where all class-files lay.

class AddCmd implements Command {
    public void cmdProc(Interp interp, TclObject[] objv) throws TclException {
        if (objv.length != 4) {
            // My add function takes 3 parameters
            throw new TclNumArgsException(interp, 1, objv, "arg1 arg2 arg3");
        } else {
            ...
            int result = TclInteger.get(interp, objv[1]) + TclInteger.get(interp, objv[2]) + TclInteger.get(interp, objv[3]);
            interp.setResult(result);
        }
    }
}

public class addTCLCommand extends Extension {
    public void init(Interp interp) {
        interp.createCommand("MyAdd", new AddCmd());
    }
}

The Script to use this command
package require java
java::load addTCLCommand

MyAdd 2 3 4 #returns the value of (2+3+4)

A.1.6 Variable passing between scripts

This example shows how variables can be passed between scripts

Script 1
global globalVariable
set globalVariable "string1"

Script 2
puts $globalVariable # prints “string1” if script 1 are executed before this, otherwise an error occurs
set globalVariable "string2" #now the global variable contains the string “string2”

proc functionCall_1 {} {
    puts $globalVariable
}
proc functionCall_2 { globalVariable } {
    puts $globalVariable
}

functionCall_1 # Error, global variable cant be accessed in a “function"
functionCall_2 $globalVariable # Works
A.2 JavaScript (Rhino)

A.2.1 Syntax description
Since the syntax does not differ much from real Java code are only a few things worth to mention here, i.e. the things that differs.

functionName(…)
\[function\ functionName(…)
String aString
\[var\ aString
int anInt = 8
\[var\ anInt = 8
myClass.staticFunction(…)
\[Packages.myClass.staticFunction(…)

A.2.2 Creating a simple GUI
This example will show how to create a simple GUI from a script.

var frame = new Packages.javax.swing.JFrame;
var button = new Packages.javax.swing.JButton("Press me");
var contentPane = frame.getContentPane();
contentPane.setLayout(new java.awt.GridLayout(3, 2));
contentPane.add(button);
function rhino_button_event() { print("hello");}
var o = { actionPerformed: rhino_button_event }
var buttonListener = java.awt.event.ActionListener(o);
button.addActionListener(buttonListener);
frame.show();

The script could either be executed via the shell like environment Rhinos has, or through Java code, described in A.2.4. The shell like environment uses the Java interpreter together with the jar-file included in the release of Rhino and is accessed through the command: “java org.mozilla.javascript.tools.shell.Main <scriptFile>”, where the “scriptFile” is optional.

A.2.3 Scripting a program modification
This examples shows how Rhino could exchange an applications execution path, i.e. how to exchange an application event without destroying the original application.

Java class
public class EventClass extends JFrame {
…
public JButton jButton1 = new JButton("Press me");
//add the button to the frame and make a event for it.
…

public static void main(String[] args) {
…
}
}

The script
var eventFrame = new Packages.EventClass(); #EventClass exists in CLASSPATH
var button1_actions = (eventFrame jButton1.getListeners(java.awt.event.ActionListener));
Like shown in appendix A.1.3 a button substitution could also be done with Rhino but there is no point of showing the code for that since it is just like doing it in pure Java code.

### A.2.4 Execute a script from Java code

This example shows how a script-file could be used from inside Java code.

```java
public class RunScript {
    ...
    public execScript(String scriptFile) {
        Context cx = Context.enter();
        try {
            Scriptable scope = cx.initStandardObjects(null);
            Object result = new Object();
            Script test = cx.compileReader(scope, new java.io.FileReader(scriptFile), "script", 0, null);
            result = test.exec(cx, scope);
            ...
        } catch (JavaScriptException jse) {
            ...
        } catch (IOException ioe) {
            ...
        }
    }
    ...
}
```

Of course, one could not only execute a whole script-file from the inside of Java code, but also just one command and that code would look like this:

```java
public class ExecJavaScriptCmd {
    ...
    public execCmd(String scriptCmd) {
        Context cx = Context.enter();
        try {
            Scriptable scope = cx.initStandardObjects(null);
            Object result = new Object();
            Script test = cx.evaluateString(scope, scriptCmd, "cmd", 1, null);
            result = test.exec(cx, scope);
            ...
        } catch (JavaScriptException jse) {
            ...
        }
    }
    ...
}
```
The argument passing could be done this way. The script uses the variable names “args0” to “argsN” for access to the arguments.

```java
public class ExecJavaScriptCmd {
    ...
    public execCmd(String scriptFile, String[] args) {
        Context cx = Context.enter();
        try {
            Scriptable scope = cx.initStandardObjects(null);
            Object result = new Object();
            for(int i=0; i < args.length(); i++) {
                cx.evaluateString(scope, "args" + i + " = new java.lang.String(" + args[i] + ");", "setup", 1, null);
            }
            Script test = cx.compileReader(scope, new java.io.FileReader(scriptFile), "script", 0, null);
            result = test.exec(cx, scope);
            ...
        }
        catch (JavaScriptException jse) {
            ...
        }
    }
}
```

### A.2.5 Compiling a script-file into a class-file

A simple script-file is compiled into a class-file. This is then decompiled back into Java-code.

#### Script

```java
var str = new String("Hello from Rhino script-file");
print(str);
```

#### Decompiled class-file from the compiled script-file

```java
import org.mozilla.javascript.*;
import org.mozilla.javascript.optimizer.OptRuntime;

public class test2 extends NativeScript implements Script {
    public test2() {
    }
    public static final void main(String args[]) {
        ScriptRuntime.main(Class.forName("test2"), args);
    }
    public void initScript(Scriptable scriptable, Context context) {
        new String[] { "str" };
        this;
        JVM INSTR swap;
        argNames;
    }
    public static Object getSourcesTreeImpl() {
        return "\222{,\003stra\200\036,\006String^,\034Hello from Rhino script-file_Y\001,\005print^,\003str_Y\001";
    }
    public final Object exec(Context context, Scriptable scriptable) {
        initScript(scriptable, context);
        return call(context, scriptable, scriptable, null);
    }
    public final Object call(Context context, Scriptable scriptable, Scriptable scriptable1, Object aobj[]) {
    }
```
scriptable = ScriptRuntime.initScript(context, scriptable, this, scriptable1, false);
Object obj = Undefined.instance;
ScriptRuntime.setName(ScriptRuntime.bind(scriptable, "str"), ScriptRuntime.newObject(context,
ScriptRuntime.name(scriptable, "String"), new Object[] {
    "Hello from Rhino script-file"
}, scriptable), scriptable, "str”);
obj = OptRuntime.callSimple(context, "print", scriptable, new Object[] {
    ScriptRuntime.name(scriptable, "str"")
});
ScriptRuntime.popActivation(context);
return obj;
}

This shows that a compiled script-file could be executed with the help of an ordinary Java-
interpreter. Of course the compiled class-file could also be executed through a script. The
code for doing this is described below.

...loadClass("<compiled script-file>"); ...

**A.2.6 Variable passing between scripts**

This example shows how variables are passed between scripts.

**Script 1**
var globalVariable = new java.lang.String("string1");

**Script 2**
globalVariable = globalVariable + " string2";  #globalVariable contains “string1 string2” if script 1 are
#executed before script 2
A.3 Visual Basic

This example shows the usability of J-Integra and Visual Basic.

Java class that Visual Basic script uses

```java
class JavaInterface {

    Bin_Hex binhex = new Bin_Hex();
    public String function_1() {
        return ("something important");
    }
    public int function_2() {
        return (2);
    }
    public static void main(String[] args) throws Exception {
        com.linar.jintegra.Jvm.register("Java_Interface");
        Thread.sleep(6000000);  //sleep for a while so that Visual Basic code could access these functions
    }
}
```

Visual Basic script

```vbnet```
Dim JavaInterface As Object
Set java_interface = GetObject("Java_Interface:JavaInterface")
Dim return_string As String
return_string = java_interface.function_1()
Dim return_int As Integer
return_int = java_interface.function_2()
```

How to run this example

Example:

1) set CLASSPATH=%CLASSPATH%;<path_to_jar-file>
2) regjvmcmd DecBinInterface localhost[<port>]
3) java -DJINTEGRA_DCOM_PORT=<port> JavaInterface
4) regjvmcmd /unregister Java_Interface

Description:

1) CLASSPATH variable must contain the “jintegra.jar” file that j-integra application includes.
2) The regjvmcmd binary-file are then used to register the com-object to a specific port.
3) Start the Java-application
4) Unregister the com-object
A.4 Example of an obfuscated program
This example shows how “jode” makes the obfuscation of a simple Java-application.

Uncompiled Java files:

```
//FileProxy.java
public class FileProxy {
    private User user;
    private Password_util pass_util;
    private ACL_util acl_util;
    private boolean ValidateUser() {
        try {
            return pass_util.validateUser(user.getName(), user.getPasswd());
        } catch (Exception e) {} 
        return false;
    }
    protected FileProxy(User i_user) {
        user = i_user;
        pass_util = new Password_util();
        acl_util = new ACL_util();
    }
    public static FileProxy getFileProxy(String i_UserName, String i_Passwd) {
        User user = new User(i_UserName, i_Passwd);
        FileProxy fp = new FileProxy(user);
        if (fp.ValidateUser())
            return fp;
        else
            return null;
    }
    public String readLineFromFile(String i_FileName) {
        if (checkACL(i_FileName)) {
            //read the file
        }
    }
    private boolean checkACL(String i_filePath) {
        return acl_util.checkACL(user, i_filePath);
    }
}

//ACL_util.java
public class ACL_util {
    public boolean checkACL(User i_user, String i_filePath) {
        return true; //return true for now
    }
}
```
//User.java

public class User {
    private String userName;
    private String passwd;

    public User(String i_userName, String i_passwd) {
        userName = i_userName;
        passwd = i_passwd;
    }

    public String getName() {
        return userName;
    }

    public String getPasswd() {
        return passwd;
    }
}

//Password_util.java

public class Password_util {
    private MessageDigest md;

    public Password_util() {
        try {
            md = MessageDigest.getInstance("SHA");
        } catch (java.security.NoSuchAlgorithmException ex) {} 
    }

    public boolean validateUser(String i_userName, String i_passwd) throws Exception {
        boolean found = false;
        boolean allInFileChecked = false;

        byte[] checkWithFile = new byte[25];
        byte[] tmp = md.digest(new String(i_passwd+i_userName).getBytes());

        for (int i = 0; i < 25; i++) {
            if (i < tmp.length)
                checkWithFile[i] = tmp[i];
            else
                checkWithFile[i] = tmp[0];
        }

        FileInputStream fi = new FileInputStream("the_file");
        while(!allInFileChecked && !found) {
            byte[] filedata = new byte[25];
            tmp_int = fi.read(filedata, 0, 25);

            if(tmp_int <= 0)
                allInFileChecked = true;
            else if (md.isEqual(filedata, checkWithFile))
                found = true;
        }

        fi.close();
        return found;
    }
}

When the interface (FileProxy) is left unscrambled by “jode” the decompiled class-files looks like this:
public class FileProxy {
    private boolean a() {
        try {
            return A.a(a.a(), a.A());
        } catch(Exception _ex) {
            return false;
        }
    }

    protected FileProxy(b b1) {
        a = b1;
        A = new c();
        b = new a();
    }

    public static FileProxy getFileProxy(String s, String s1) {
        s = new b(s, s1);
        s = new FileProxy(s);
        if(s.a())
            return s;
        else
            return null;
    }

    public String readLineFromFile(String s) {
        String s1 = "error";
        if(a(s))
            try {
                s = new FileInputStream(s);
                this = new DataInputStream(s);
                this = new BufferedReader(new InputStreamReader(this));
                s1 = readLine();
                s.close();
            } catch(Exception _ex) { }
        return s1;
    }

    private boolean a(String s) {
        return b.a(a, s);
    }

    private b a;
    private c A;
    private a b;
}

public class a {
    public a() {
    }

    public boolean a(b b, String s) {
        return true;
    }
}

//b.java (User.java)

public class b {

    public b(String s, String s1) {
        a = s;
        A = s1;
    }

    public String a() {
        return a;
    }

    public String A() {
        return A;
    }

    private String a;
    private String A;
}

//c.java (Password_util.java)

public class c {

    public c() {
        try {
            a = MessageDigest.getInstance("SHA");
        } catch(NoSuchAlgorithmException _ex) { }
    }

    public boolean a(String s, String s1) throws Exception {
        boolean flag1 = false;
        boolean flag = false;
        byte abyte0[] = new byte[25];
        s1 = a.digest((new String(s1 + s)).getBytes());
        for(s = 0; s < 25; s++)
            if(s < s1.length)
                abyte0[s] = s1[s];
            else
                abyte0[s] = s1[0];
        FileInputStream fileinputstream = new FileInputStream("the_file");
        while(!flag && !flag1) {
            s1 = new byte[25];
            s = fileinputstream.read(s1, 0, 25);
            if(s <= 0)
                flag = true;
            if(MessageDigest.isEqual(s1, abyte0))
                flag1 = true;
        }
        fileinputstream.close();
        return flag1;
    }

    MessageDigest a;
}