The Inject/J Tutorial

Table of Contents

1 Introduction to Inject/J
   What is in the document
   Introduction

2 Installation

3 Using Inject/J
   Starting Inject/J
   Configuring Inject/J - The configuration dialog
   Creating a new project and setting the output path
   Choosing script sources
   Choosing Java Source-/Classpaths
   Choosing Java Source Paths/files
   The Finish dialog
   Script Execution

4 Lesson 1: Visitors
   example 1: The visitor
     How it works
     Example Session
   example 2: Visitor with direct navigation
     How it works
     Example Session
   example 3: A method call trace
     The Script
     How it works
     Example Session

5 Lesson 2: Interfaces
   example 1: The comparable interface
     The Script
     How it works
     Example Session
   example 2: The interface extractor
     How it works
     Example Session

6 Lesson 3: Design patterns
   About design patterns
   example 1: The singleton design pattern
     The script: singleton.ij
     The library: singleton.ijl
     How it works
     Example Session
   example 2: The factory design pattern
     The script
     The library: factory.ijl
     How it works
     Example Session
   example 3: Delegation
     How it works
     Example Session

7 Lesson 4: A complex example
   A complex example: visualization of classes
     The datavis library
     How it works
     Data Visualizer
     Example Session

8 Lesson 5: Adaptive Programming using Inject/J
   Introduction to Adaptive Programming
   The AP-based model of Inject/J
   Language elements of the AP extension
   Example 1: Hello World in Adaptive Programming
Chapter 1. Introduction to Inject/J

Table of Contents

What is in the document
Introduction

What is in the document

The tutorial consists of the following parts:

1. Introduction: Explains the motivations behind the development of Inject/J and the key concepts of the system.
2. Installation Guide: Explains the installation sequence. You will find Inject/J very easy to setup.
3. GUI tutorial: Explains how to use the Graphical User Interface.
4. Lesson 1: The first practical examples of Inject/J. You will learn the basic concepts, and understand some simple scripts.
5. Lesson 2: The first really usable examples. It describes how to implement interfaces automatically using Inject/J.
6. Lesson 3: Describes some frequently used design patterns and their automatic implementation using Inject/J.
7. Lesson 4: A complex example that generates simple visualization for the state of a class.
8. Feature Matrix: Lists the Inject/J language features and links to the examples where they are explained in the tutorial.

Introduction

Software development is an evolutionary process. Requirements, which are unclear and so misunderstood, or the new demands the software has to meet, lead to major (or minor) changes of design and implementation. Examples of these requirements and their consequences:

- Flexibility: due to the lack of flexibility, the system cannot be extended, and therefore proper reconstruction is needed
- Performance: superfluous flexibility leads to performance loss. Therefore sometimes we need to remove unnecessary flexibility and the correlating indirections.
- Structural and interface requirements: by the reuse of existing software components it often happens, that the given interfaces do not pass to the required interfaces. We could mention the version changes, or the changes of libraries as example (e.g. different versions of JDK)
- Persistence and data model: during development or reuse of software, the data model and the way of storing data change very often
- Composition and protocol requirements: interaction, communication and synchronization change very often during development
- Externalization of the program run: in different phases of development, there are different requirements for error-handling, tracing, profiling, debugging, etc.
If these requirements change, the software must be modified. This process is called software adaptation. Software adaptation is a demanding task, due to the fact that the above mentioned requirements cannot be met by using the standard facilities of existing programming languages (e.g. encapsulation, inheritance). Instead of these, modifications of the whole internal implementation structure are necessary. Even if only local modifications are needed, the manual implementation is exposed to errors.

Inject/J is a tool for automated code transformation for Java programs. It consists of a metamodel for automated code transformation, a simple script language to describe the transformation operations that are to be executed, a script processor and a metaprogramming library (RECODER) for the eventual program transformations. The Inject/J transformation language is based on four concepts:

1. Namespace: involves classes as starting point of the navigation.
2. Weaving points: they define specific places in the system, where the so-called weaving operations can be carried out.
   a. Implicit weaving points are defined by the language metamodel: class, method, attribute, method call, attribute access, attribute assignment, declaration of local variable and access/assignment of local variable.
   b. Explicit weaving points mark those places of possible future adaptions, which cannot be described by the language metamodel
3. Navigation: The User navigates to weaving points, in order to use transformations on them.
   a. Hierarchical Navigation: The navigation happens through the Language Metamodell. For example: In order to access the methods of a class, one has to navigate to the class in advance.
   b. Direct Navigation: The navigation happens directly to a previously visited weaving point, by using its reference.
4. Transformations define operations to be used in weaving points. The available types of transformations are: before, after, add, change, replace, delete.

Chapter 2. Installation

1. Download either the file InjectJFullInstall.jar or the file InjectJBaseInstall.jar from http://injectj.fzi.de
2. Change directory to the one you saved the file to, and type
   a. java -jar InjectJFullinstall.jar or
   b. java -jar InjectJBaseInstall.jar respectively.
3. This will start the installation program of Inject/J. During the installation process you will be presented with a number of screens. You can proceed to the next screen by pressing the “NEXT>>” button, or you can go back to the previous screen by pressing the “<<PREVIOUS” button. You can quit the installation process anytime by pressing the “QUIT” button. After starting the program, the following screen will appear:

Figure 2.1. Language selection

![Language selection](image)

4. Choose the language and press the “OK” button
5. After reading the welcome message proceed to the next screen, by pressing the “NEXT” button.

6. Read carefully the Licence Agreement, choose “YES” if you agree and proceed to the next screen. If you choose “NO”, you will not be able to install the program.

7. Choose the installation path. You can either type in the destination folder of the installation, or choose it in a dialog.

   **Figure 2.2. Selecting Installation Path**

![Select the installation path image](image)

8. Choose the packages you would like to install.

   **Figure 2.3. Choosing Packages**
9. Launch the installation process by pressing the “INSTALL” button

10. The installation has finished. If you run Inject/J under a Unix/Linux system,
    a. go to the directory `[installation directory of Inject/J]/bin`
    
    b. type `chmod +x StartInjectJ`

**Chapter 3. Using Inject/J**

Table of Contents

StartInjectJ
Configuring Inject/J - The configuration dialog
Creating a new project and setting the output path
Choosing script sources
Choosing Java Source/Classpaths
Choosing Java Source Paths/files
The Finish dialog
Script Execution

**Starting Inject/J**

- *Under a Unix/Linux System*

  Run the `StartInjectJ` script, which is located in the `[Inject/J installdir]/bin` directory, 
  e.g.: `/home/brian/InjectJ/bin/StartInjectJ`
Under Windows

Run StartInjectJ.bat, which is located in the \[Inject/J installdir]\bin directory, e.g.: \c:\Inject/J\bin\StartInjectJ.bat

After starting Inject/J two windows appear. The Inject/J Output window shows the output of Inject/J to the standard out and standard error. This plays an important role when you execute an Inject/J script.

The main window of Inject/J shows a welcome message at startup. Clicking on the “ABOUT” button brings up a dialog with copyright information. You can navigate through all windows with the “NEXT>>” and “<<PREVIOUS” buttons. For help, click the “HELP” button. Clicking the “QUIT” button will terminate the application. Pressing the “CONFIGURATION” button will bring up the Configuration dialog.

Figure 3.1. Inject/J Output Window

Figure 3.2. The Welcome Screen
Configuring Inject/J - The configuration dialog

This dialog allows the user to configure the appearance and behaviour of Inject/J. The following selections can be made:

- **Look&Feel**

  Different Look&Feels can be chosen, depending on the installed JVM. All available Look&Feels are detected automatically, and the default one is Metal.

- **Language**

  Choose the language here. New languages can be added by editing the `injectj.ini` file manually.

- **Project Base Directory**

  This text field contains the path to the Inject/J default project files directory. Inject/J project files must not necessarily reside in this directory (or in any of its subdirectories), they can be located anywhere. This path simply specifies the starting point for the project file chooser dialog.

- **Script Base Directory**

  This path specifies the default base directory to Inject/J’s script source files. Similarly to the project files, these can also be located anywhere in the file system. This path only configures the script file chooser dialog to start at the given location.
• **Java Base Directory**

  The default base directory for the Java source dialog.

• **Library Base Directory**

  The directory, in which Inject/J searches for libraries. The filenames of these libraries must be the same as in the library declaration, ending with ".ijl".

• **Default Java libraries**

  Specifies the default Java libraries/paths to be used with every Inject/J project. At least the path to the Java runtime library "rt.jar" should be set. You can add new libraries by specifying their path in the text field and pressing the "ADD" button, or you can use the "..." button to invoke the file system browser for choosing the path. To remove a library/path from the list, select it, and press the "REMOVE" button.

• **Ignore upper/lower case in path specifiers**

  You should check in this box, if you have a case insensitive operating system (e.g. Windows 95/98/NT/2000/XP), but if not, please uncheck it (e.g. Unix/Linux)

• **Save**

  Save the current settings to "injectj.ini" (the configuration file of Inject/J). Saving the settings automatically applies them.

• **Apply**

  Apply the current settings without saving them.

• **Cancel**

  Closes the configuration dialog and restores all settings to their original state.

• **OK**

  Applies the current settings, and closes the configuration dialog, but the new settings will not be saved. If you want them to be saved, please use the "SAVE" button first.

**Figure 3.3. The Configuration Dialog**
Creating a new project and setting the output path

Press the “NEXT>>” button on the welcome screen, this will get you to the “New project” dialog. Enter the project file with absolute path: this can be an existing, or a new project file. You can use the file system browser to enter the path conveniently by pressing the “...” button. An Inject/J project file has the extension “.ijp”. In this tutorial we create a project file “test.ijp”, so enter:

(a) `your home/InjectJ/Examples/Projects/test.ijp` - UNIX/LINUX [1]

(b) `c:\InjectJ\Examples\Projects\test.ijp` - Windows [2]

Java Output Path specifies the directory in which Inject/J puts transformed source. Enter

(a) `your home/InjectJ/Examples/Output` - UNIX/LINUX

(b) `c:\InjectJ\Examples\Output` - Windows

Now proceed to the next dialog by pressing “Next>>”

Figure 3.4. The New Project Dialog
Choosing script sources

You can add script source files either using the file system browser (button “...”) or by specifying them directly. Script source files have the extension “.ij”. After pressing the “ADD” button the specified script source appears in the “Script source files” list. These scripts will be executed later. With the “Move Up”/”Move Down” buttons you can specify the order, in which the script sources are parsed afterwards. The order is important, if you use macro libraries or macros, which are used in more than one script file. As macros are only available after their definition, the macro-defining files must be parsed first. This has no effect on the execution order! Pressing the “Remove” button removes the selected script file(s).

Now add

(a) /your home]/InjectJ/Examples/Scripts/Observifier.ij

(b) c:\InjectJ\Examples\Scripts\Observifier.ij

Note: if you add more than one script source, you will get an extra dialog after pressing "Next" to specify which script(s) to execute and in what order. Since it is possible that a script needs the modifications done by another script, the execution order is important.

Now proceed to the next dialog by pressing “Next>>”

Figure 3.5. Choosing Script Sources
Choosing Java Source-/Classpaths

Using this dialog you can choose the source-/classpath(s) of the Java source/classfile(s) to be processed by the script(s). You should specify the root of your source directories here in order to keep the proper package structure.

Do not forget that you also have to specify all the additional libraries that the sources need, either by locating the "zip", "jar" files, or by giving the path to the necessary "class" files.

In this tutorial we process the package "observifier". This package resides in "~/InjectJ/Examples/Sources", which is the root of the sources, so enter

(a) [your home]/InjectJ/Examples/Sources

(b) c:/InjectJ/Examples/Sources

Now proceed to the next dialog by pressing “Next>>”

Figure 3.6. Choosing Java Source-/Classpaths
Choosing Java Source Paths/files

Select the source files/packages that you want to transform from the list of possible source files. Then by clicking on the "Add subtree(s)" button they will be added to the selected source files list. Since we want to transform the package "observifier", add it to the selected sources list. As we do not want to configure the namespace of Inject/J, leave the option “Use all classes as Namespace” ticked in. However, if you untick this option, you get an extra dialog to configure the namespace. Now proceed to the next dialog by pressing “Next>>”

Figure 3.7. Choosing Java Source Paths/Files
The Finish dialog

This dialog shows you the current settings. If you select the “Trace navigation” option, Inject/J will print extra trace information - this can be useful for debug purposes. You can save the project by pressing the “Save project” button. By pressing the “Finish” button all selected scripts will be executed on all specified sources. If you have not saved your project yet, Inject/J will ask you if you wanted to do so.

Figure 3.8. The Finish Dialog
Script Execution

You can follow the standard output of the scripts in the “Inject/J output” window.

During the transformation you should specify "observifier.SubjectTest" as subject, and "observifier.ObserverTest1","observifier.ObserverTest2" as Observer.

[1] we assume that you installed Inject/J to ~/InjectJ

[2] we assume that you installed Inject/J to c:\InjectJ

Chapter 4. Lesson 1: Visitors

Table of Contents

element 1: The visitor
   How it works
      Example Session

element 2: Visitor with direct navigation
   How it works
      Example Session
example 3: A method call trace

The Script

How it works

Example Session

After completing this lesson you will understand:

- The basic navigation concepts of Inject/J
- How a weaving point can be used
- How to interact with the user

example 1: The visitor

The following script visits all classes and all methods in the namespace, and prints out their name.

```inject/j
script visitor {
    foreach class '*' <=c> in namespace do {
        console("Class: "+c.name);
        console("Methods:");
        foreach method '*(*)' <=m> do {
            console(" "+m.signature);
        }
        console("Constructors:");
        foreach method '*(*)' <=m> in c.constructors do {
            console(" "+m.signature);
        }
        console("\n");
    }
}
```

How it works

script visitor {

An Inject/J script begins with a script declaration. Every script has a name. It is possible to define several scripts in one script file.

```
foreach class '*' <=c> in namespace do {
```

The foreach statement navigates through the namespace, and matches all classes. As you might notice, this is the implementation of the Hierarchical Navigation concept. However, the statement can operate on any list like structure. The namespace variable is implicit (always defined) and it is actually a list of classes. If the `in <variable>` part is omitted, the list of weavingpoints is acquired from the current context. The context defines the list of weavingpoints that can be reached directly from the actual position in the hierarchy. From a class context one can navigate to the methods and attributes of the class. In the global context only the class weavingpoints of the namespace are reachable. The statement works very much like the foreach statement of other programming languages.

**Syntax:** `foreach <weavingpoint type> <string identifier> [/<variable reference>] [in <list>] do { <statements> }
```

- **Weavingpoint type:** Specifies the weavingpoint type. For example: class, method, access.

- **String identifier:**
  - ambiguous identifier: The identifier matches more than one weavingpoint. For example: 'test.example.*'
  - non-ambiguous identifier: The identifier matches exactly one weavingpoint. In this case the statement works much like direct navigation. For example: 'test.example.oneclass'
Variable reference: With the `<=variable>` syntax, it is possible to reference the actual weavingpoint with a name. This is the local variable concept of the Inject/J language. The variable is only defined in the statement body. The variable type is determined from the navigation context.

List: It is a list type variable. If omitted, it defaults to namespace.

The body of the statement will be executed for each class found in the namespace, as the script visits the weavingpoint.

```javascript
console("Class: "+c.name);
console("Methods:");
```

The console statement prints its argument to the standard output. It is possible to concatenate strings. The variable identifier notation represents access to a weavingpoint's attribute. Every weavingpoint type has different attributes.

```javascript
foreach method '*(*)' <=m> do {
  console("  "+m.signature);
}
```

This statement iterates through the methods of the class weavingpoint. As described earlier, the statement acquires the method list from the actual context.

```javascript
console("  "+m.signature);
```

The script prints the method's signature.

```javascript
}
  console("Constructors:");
  foreach method '*(*)' <=m> in c.constructors do {
    console("  "+m.signature);
  }
}
```

But in order to navigate to the constructors of the class only, one has to explicitly specify the list of the constructors. The list of constructors is defined as a property of the class weavingpoint.

```javascript
console(""); And do some cosmetics.
}
```

**Example Session**

If we run the script on a `TestClass`, we get the following output:

```
Class: TestClass
Methods:
  void hiddenMethod()
  java.lang.String getString()
  java.lang.String toString()
  TestClass()
  void main(java.lang.String[])
  TestClass(java.lang.String)
  void doSomething()
  void setString(java.lang.String)
  TestClass(int)
Constructors:
  TestClass()
```
example 2: Visitor with direct navigation

The following script has been taken from the Inject/J Language Specification.

class : aClass;
list(method) : methodList;
script AskExample {
    ask("Please select a class to visit", namespace, aClass);
    in class aClass do {
        console("Visiting class '"+aClass.name+"'");
        ask("Please select some methods to visit", aClass.methods, methodList);
        foreach method '*(*)' <=m> in methodList do {
            console("Visiting method '<m.signature>'");
        }
    }
}

How it works

class : aClass;
list(method) : methodList;

It is possible to declare global variables in Inject/J script files. These declarations are file-global, therefore they can be used in every script defined in the file. The following variable types are available:

- Primitive types: string, integer, bool
- Weavingpoint types: class, attribute, method, access, assignment, return, exception, exception throw, local variable declaration, explicit, parameter
- List types: lists can be built from any of the types above.

script AskExample {

    ask("Please select a class to visit", namespace, aClass);

Script declaration.

    in class aClass do {

Intelligent user interaction. This statement pops up a dialog box. The contents of the dialog box depend on the type of the parameters. The first argument is always the message shown to the user. When called with 2 parameters, it is possible to ask for simple values, such as integer, string and bool. If the result variable was initialised before, its value is shown as default value. Example: ask("Continue?", boolVar)

If called with 3 parameters, the second parameter must be a list of weavingpoints of the same type. This way the user can choose one or more weavingpoints from the list, depending on the type of the third parameter. Of course the third parameter's type has to be compatible with that of the second parameter.

    in class aClass do {

Direct navigation. The parameter has to be an initialised variable of the specified type. The navigation context is set to the specified weavingpoint, and the statement body is executed. After executing the statement body, the navigation context is restored.
console("Visiting class "+aClass.name+" ");

Console.

ask("Please select some methods to visit", aClass.methods, methodList);

User input.

foreach method '(*)(*)' <=m> in methodList do {

Navigate to the selected methods.

console("Visiting method '<m.signature> ");

Console message. This statement shows the variable embedding method in strings.

}

}

Example Session

If we run the script on a TestClass, we get similar output:

Visiting class 'TestClass'
Visiting method 'TestClass()'
Visiting method 'TestClass(java.lang.String)'
Visiting method 'setString(java.lang.String)'

dexample 3: A method call trace

For example a method

public static void foo() {
  do something....
}

should be transformed into:

public static void foo() {
  System.out.println("Classname: void foo()");
  do something....
}

The Script

declare list(class): classes;
script CallTrace {
  ask("Please select the classes to transform!",namespace,classes);
  foreach class '(*)(*)' <=c> in classes do {
    foreach method '(*)(*)' <=m> do {
      before ${
        System.out.println("<c.name>:<m.signature> ");
      }$;
    }
  }
}
How it works

The only new statement is `before`. This is a code transformation (weaving) statement. It can be used in a weavingpoint, to insert code before the weavingpoint. For methods this means, that you can insert code in the method body before the other statements. There are two other weaving statements in the Inject/J languages similar to before:

- `after`: Inserts the code specified in its body after the weavingpoint.
- `replace`: Replaces the sourcecode corresponding to the actual weavingpoint with the code specified in its body.

The weavingpoint types interpret the `after` and `before` statements differently. For example `after` for a method inserts the new code at each non-error method exit point (as a rule thumb: just before each return statement). The full specification can be found in the Inject/J Language Specification.

Example Session

If we run the script on a `TestClass`, we get the transformed source.

Chapter 5. Lesson 2: Interfaces

Table of Contents

- example 1: The comparable interface
  - The Script
  - How it works
  - Example Session
- example 2: The interface extractor
  - How it works
  - Example Session

After completing this lesson you will understand:

- Dealing with interfaces
- Different ways of user interaction in Inject/J: console / message / ask
- Branch statement: if
- Using variable properties
- Creating new classes

example 1: The comparable interface

You often have classes you wish to compare. In order to do this in a standard way, you have to implement the `Comparable` interface of the Java Runtime. The following script makes it possible to implement the necessary methods for the interface automatically.

The Comparable interface contains one method: public int compareTo(Object o). This method is added to the original class.

```java
class C implements Comparable {
    ...
}
public int compareTo(Object o) {
    ...
    ...
}

The Script

The script can be found here: MakeComparable.ij

How it works

declare class : aClass;
declare attribute : cmpAttribute;

Declare variables.

script MakeComparable {
    ask("Please select a class to make it comparable", namespace, aClass);

    Choose the class to transform.

    in class aClass do {

        Navigate to the selected class.

        console("Processing class "+aClass.name);

        Print message to the standard output.

        if ( aClass.attributes.isEmpty ) do {
            message("Class has no attributes. What should I compare?");
            exit(0);
        }

        Check whether the class has any attributes. If not, the message() statement pops up a message box, and exits. The parameter of the exit() call specifies the return value to the operating system.

        ask("Please select the attribute to compare",aClass.attributes,cmpAttribute);

        If the chosen class has attributes, we choose one. This attribute will be compared in the compareTo() method that the script will implement.

        if (!cmpAttribute.hasPrimitiveType) do {
            message("Only primitive types are supported...");
            exit(0);
        }

        This script only supports comparing attributes of primitive types, which are: boolean, char, byte, short, int, long, float and double.

        add to implements ${ java.lang.Comparable }$;
This statement modifies the implemented interfaces of a class. The interfaces to be implemented must be specified with a comma separated list. Fully qualified names have to be used, or the needed packages have to be imported using the "add to imports" command. It adds the newly specified interfaces to the implementation list.

\[
\text{if (cmpAttribute.staticType.name.equals("boolean") } \) do { }
\]

If cmpAttribute's type is boolean, we have to create a different comparison function. “staticType” is an attribute of the “attribute” weavingpoint type, which holds the type of the attribute, in form of a class reference.

\[
\text{add to members \{ }
\begin{align*}
\text{public int compareTo(Object o) \{ }
\text{<aClass.name> x = (<aClass.name>) o; }
\text{if (cmpAttribute.name &amp; x.<cmpAttribute.name>) \{ }
\text{return 0;}
\text{\} else \{ }
\text{return 1;}
\text{\}
\text{\} }
\end{align*}
\]

This statement adds the statements in its body to the class definition. In this case it adds a method to it. Note, how the values of Inject/J variables are inserted in the Java code, using the \text{<weavingpoint.attribute>} syntax. The attribute can only have simple type.

\[
\text{}}$;
\]

In the other case, the script includes another comparison function designed for numeric types.

\[
\text{}}$;
\]

\[
\text{}}$
\]

Example Session

If we run the script on a TestClass, we get a transformed class similar to this one.

\[
\text{example 2: The interface extractor}
\]

Sometimes we would like to allow accessing the functions of our classes, but do not want to allow the whole functionality of the class to be seen. This is very typical, if we want to add a script language (e.g. Jython) to our commercial Java program, but do not want to allow the script to call constructors, etc.

Using interfaces would be a proper solution, but during the programming process we do not always have time to create interfaces for each class we use. This problem can be easily solved by an Inject/J script!

The following script takes a class Foo and creates an interface named IFoo from the public methods in Foo. The modified source would contain:

\[
\text{class Foo implements IFoo } \{ \\
\text{\} }
\text{interface IFoo } \{ \\
\]
...the public method signatures of Foo...
"

The whole script can be found here: InterfaceExtractor.jj

How it works

declare class : aClass;
declare string : ifName;
declare string : oldName;
declare string : newName;
declare string : methodDef;
declare integer : currentParam;

Declare some variables.

script InterfaceExtractor {
    ask("Please select a class to extract the interface from", namespace, aClass);

Choose the class to transform.

    in class aClass do {

Navigate to the chosen class.

        ifName = aClass.name+$END$;
        oldName = aClass.simpleName+$END$;
        newName = "I"+aClass.simpleName;
        ifName=ifName.replaceSubstring(oldName,newName);

Create the name of the interface. The string type has some simple string manipulation functions. The full function list is described in the Inject/J Language Specification.

        embed {

The embed statement is very useful, when one has to temporarily change the navigation context, but needs access to the already defined local variables. The statements in the body are executed in the default (namespace) context, but the local variables are available. In this case one needs to use embed because classes and interfaces can only be defined in the global (default) context.

        add interface ifName ${
    }

Adds an interface to the namespace. A valid interface definition body must be included in the body. This example creates an empty interface.

    } foreach method '*(*)' <=m> in aClass.methods do {
        if (!m.isConstructor & !m.isImplicitConstructor &
            m.isPublic & !m.isFinal & !m.isStatic) do {

For every non-constructor method of the class, the script builds the method definition using the method weavingpoint's attributes. Only public methods can be exported into an interface.

            methodDef="";
            if (m.isPublic) do {

                

            }
methodDef=methodDef+" public ";
if (m.isAbstract) do {
    methodDef=methodDef+" abstract ";
} if (m.isSynchronized) do {
    methodDef=methodDef+" synchronized ";
}

Insert modifier keywords. Only public and abstract modifiers are allowed in an interface definition.

methodDef=methodDef+m.returnType.name;

Specify return type.

methodDef=methodDef+" ";
methodDef=methodDef+m.name;
methodDef=methodDef+"(";

Specify method name.

iterate {currentParam,m.parameters} do {

The iterate statement can be used for iterating through the elements of a list. The first parameter is an integer variable, which holds the current iteration value. The second parameter is a list.

methodDef=methodDef+m.parameters.getElement(currentParam).staticType.name;
methodDef=methodDef+m.parameters.getElement(currentParam).name;
methodDef=methodDef+" ";

The statement body appends the "<parameter type> <name>," string to the methodDef variable for each parameter.

methodDef = methodDef+"$END$";
methodDef = methodDef.replaceSubstring("","$END$".DESCEND$);
methodDef = methodDef.replaceSubstring("$END$","$END$ DESCEND$" DESCEND$);
methodDef = methodDef+"$END$" DESCEND$);

Some string handling magic to delete the trailing coma and finish the method definiton.

embed {
    in class ifName do {
        add to members ${
            <methodDef>;
        }$;
    }

The statement adds members to a class definition. The statement body must contain valid java code, like attribute and method declarations.

}
add to implements ${<ifName>}$;

The original class of course implements the newly created interface.
Example Session

If we run the script on a `TestClass`, we get the `ITestClass` interface, and a new `TestClass` which implements the generated interface.

Chapter 6. Lesson 3: Design patterns

Table of Contents

- About design patterns
- example 1: The singleton design pattern
  - The script: singleton.ij
  - The library: singleton.ijl
  - How it works
  - Example Session
- example 2: The factory design pattern
  - The script
  - The library: factory.ijl
  - How it works
  - Example Session
- example 3: Delegation
  - How it works
  - Example Session

After completing this lesson you will understand:

- How to manipulate classes in an invasive way
- How to change modifiers of a weaving-point (static, public, final)
- The advantage of using a script to implement some design patterns
- How to use libraries for code reuse

About design patterns

When we talk about object-orientation, we always mention code reusability. But, when talking about reuse, we mostly deal with reusing our components, classes on the implementation level by delegation of functionality. (Inheritance implements a special way of delegation.)

Design patterns, however, allow us to reuse not only our code, but also the design of the software. They provide us some “recipes” for solving common problems in the software design process. These patterns shall be applied systematically to our concrete classes.

Applying these patterns is done mostly manually: we have to write the code, instead of having a tool, to do this well defined transformation for us. Now, we are going to see, that Inject/J is able to perform these transformations automatically. This might prove to be very useful in everyday life.

example 1: The singleton design pattern

First, let’s start with something pretty simple. Sometimes we have to create classes, which can only have one instance at a time. These so-called singletons are very useful for creating tool-, or helper-classes, which are typically large, but used by many other objects.
A singleton has no public constructor, but a static method, which can be used, either for creating a new object of the class, or (if the instance already exists) for getting reference to the only instance.

Supposing our class has exactly one default constructor, here is what we need to do to turn a class named Foo into a singleton:

- Change the constructor to private - this means we can't instantiate the class from outside anymore
- Create a private static Foo object as an attribute in the class. This will hold an internal reference to the only object instantiated. Default value shall be null.
- Insert a public, static method named theFoo(), which checks, whether the object has already been created, and either returns the reference to this object, or creates an object, and stores the reference.
- Find every calls to the default constructor and replace every call with the theFoo() method. Unfortunately Inject/J currently doesn't support this operation.

The script: singleton.ij

use library singleton;
list(class) : singletons;

script singletonize {
    // Ask for classes, which need to be singletonized
    ask("Please select the classes which shall be made a singleton."
        , namespace, singletons);
    foreach class '*' <=c> in singletons do {
        singleton.makeSingleton(c);
    }
}

As you may have noticed, this script has become very short, and singletonization is reduced to one function. Unfortunately, this is not because Inject/J is so powerful, but because we have implemented the singletonization in an external library.

Libraries allow us to place functions in an external file. These can be much more easily reused later than the scripts themselves. It is always suggestable to place reusable functionality inside libraries. A library can contain a number of functions, functions can have return types and parameters as well.

The library: singleton.ijl

This is the code of the library, which is required by the above script: singleton.ijl

How it works

The script itself asks for the classes to singletonize (from the classes in the namespace), and calls the library function with them. To be able to use the library, we have to add the following line to our code:

use library singleton;

To invoke a function in the library, we can use libraryname.functionname(parameters), such as:

singleton.makeSingleton(c);

The most interesting part is now in the library, so we shall take a closer look at it:
library singleton {

Each library must begin with the declaration of the name of the library. The library must be placed inside a file with the
same name, and extension .ijl .

public void makeSingleton(class c) {

This is a typical function in a library. Here - as in other languages - the visibility, return type and parameters of a
function can be defined.

In this function we will implement the singletonization.

    integer: globalCountVar;
    bool: hasDefault;
    bool: defaultOnly;
    method: defaultConstructor;

The variables we will use for storing the default constructor, and for deciding whether there is any other constructor
besides the default.

    in class c do {

We will perform operations on the class c, which we got as a parameter. Firstly, we get the values for the two boolean
expressions:

    foreach method '(*(*)' <*m> in c.constructors do {
        if (m.parameters.size == 0) do {
            hasDefault = true;
            defaultConstructor = m;
        } else {
            if (m.isPublic) do defaultOnly = false;
        }
    }

We have checked, whether there is a default constructor (constructor which has no parameters), and whether there is any
other public constructor. To achieve this, we have examined all the constructor methods.

As mentioned above, we need a default constructor and no other constructors

    if (hasDefault & defaultOnly) do {
        ...  
        in method defaultConstructor do {
            change modifier ${ private !static }$;
        }
    }

Here, we change the modifiers of the default constructor. With the change modifier statement the visibility of any class
member can be changed. To set values, we can use boolean-like expressions. The expression above sets private, and
unsets static modifier. (public and private attributes are automatically unset because of the private keyword, but other
modifiers, like synchronized are untouched.)

After the change, we insert the private attribute and the return method.

    in class c do {
        add to members ${
            private static <c.name> my<c.name> = null;
        }
        public static <c.name> the<c.name>() {
            if (my<c.name> == null) my<c.name> = new <c.name>();
        }
    }
This is the code, which inserts the modifications. It declares an attribute, and a method, which checks, if the attribute is null, and if so, creates a new instance and assigns it to the variable.

We are basically done. In the end, we send some message, if the singletonization failed.

```javascript
} else {
    // We were unable to perform the singletonization
    if (!hasDefault) do
        console("NO. There is no default constructor in the class.");
    if (!defaultOnly) do
        console("NO. There is more than one constructor in the class.");
}
```

**Example Session**

If we run the script on a `HelloInjectJ` class, we get the singletonized source.

### example 2: The factory design pattern

The factory design pattern is also widely used, and has many variants. We consider here the well known problem of constructors, namely that calls to public constructors cannot be prohibited easily. For example if we need to check a condition before allowing the creation of the object, or if we want to perform something, which is not possible inside the constructor.

On the other hand, sometimes we want to assure portability, and force the user to use a method for obtaining an object, instead of calling the constructor. This way, we have the possibility of returning a compatible class (or an already existing object, like the singleton), which would be impossible to achieve when using a constructor.

The solution is using factories in both cases. A factory is a method, which creates a (new) instance of a class. To stop the usage of the constructors, we disable them by changing their visibility to private. We provide the factory methods instead, which are static, and shall have the same signature as the original constructors.

Summarized, we need to do the following:

- Change all constructors in the class to private.
- For each constructor create a method named create() which accepts the same arguments as the constructor, and has a return value of type Foo.
- In this method create a new object using the constructor with the arguments received, and return the created object. Other additions may be also made in this method.
- Find every calls to the constructors and replace every call with the corresponding create() method. Unfortunately Inject/J currently doesn't support this operation.

We use libraries in the implementation again.

**The script**

```javascript
use library factory;
list(class) : factories;
script factorize {
```
/ Ask for classes, which need to be factorized
ask("Please select the classes which shall be factorized."
    ,namespace,factories);

foreach class '*' <=c> in factories do {
    factory.makeFactory(c);
}

The script has not changed much, compared to the previous script. We only have renamed the used library and invoke the
method makeFactory, instead of makeSingleton.

The library: factory.ijl
examples/Libraries/factory.ijl

How it works

library factory {

We have now created another library in an other file, this library has also one function for creating factories.

    public void makeFactory(class c) {
        string : paramList = ""; // Parameter list for each constructor
        string : paramPass = ""; // Passed parameters for the constructor in the create
        string : paramName;
        string : paramType;
        integer: globalCountVar;

We declare all variables necessary to construct the call to the factories. The variable paramList shall contain the
parameter signature of a constructor, while paramPass will only have the variables passed to the constructor.

For a constructor, for example:

    public void Foo(int a, int b, boolean bl) {...}

these variables will contain: “int a, int b, boolean bl”, and “a,b,bl”.

We have also declared a global counter variable.

The rest of the code is the factory-creation:

    in class c do {
        ...
        foreach method '*(*)' <=m> in c.constructors do {
            if (m.isPublic) do {

We will iterate through all public constructors, change them to private, and create the factory method for them.

            paramList = "";
            paramPass = "";
            ...
            in method m do {
                change modifier ${ private }$;
            }

This is the code used for changing the constructor to private access.
In the rest, we will make the create method. To have it done, we need to construct the parameters signature of the constructor, as well as the parameter list to pass over. This is done by iterating through the parameters of the constructor.

```java
iterate(globalCountVar, m.parameters) {
    if (globalCountVar > 0) do {
        paramPass = paramPass + ",";
        paramList = paramList + ",";
    }
    paramName = m.parameters.getElement(globalCountVar).name;
    paramType = m.parameters.getElement(globalCountVar).staticType.name;
    paramPass = paramPass + paramName;
    paramList = paramList + paramType + " " + paramName;
}
```

At the end of this part we have the paramList and paramPass variables containing the information needed to create the method.

Note, that we have not used the usual foreach keyword, but the iterate statement. There are two reasons, why we chose this solution:

1. The foreach statement does not support iterating through parameter-type weaving points. This is because foreach is used for navigating through the code, not through a list!

2. Even if there was a foreach, we would have troubles using it. When creating a signature, we have to place commas between the two members of the list, but we may not have a comma before the first (or after the last) member. This would be impossible to decide, when using a foreach-like iterator. But since iterate uses a global variable to count the elements, we can decide whether to place a comma, or not.

Note:

You should remember, that although there is much similarity between foreach and iterate, they are different: while foreach is used to iterate throughout weaving points, iterate can iterate through lists of any type, without affecting the current weaving point.

Having successfully understood iteration, and the code above, all we have to do, is embed the desired code into the class:

```java
in class c do {
    add to members ${
        public static <c.name> create(<paramList>) {
            <c.name> factory = new <c.name>(<paramPass>); return factory;
        }
    }$;
}
```

The inserted code will contain the factory method.

Note:

You can modify this part, to suite you needs better, for example you can write some extra code before the new statement.

**Example Session**

If we run the script on a **TestClass**, we get the transformed source.

**example 3: Delegation**

In this example, we try to demonstrate how to reuse transformations in Inject/J.
You should be familiar with the concept of delegation from object-oriented programming. It is a method for dynamically extending an object, providing the same interface the delegate has.

The next library function we discuss is able to perform the delegation of a delegate class into a delegating class:

- create a private variable myDelegate
- for each public method foo() of the delegate, create a method foo(), which invokes the operation myDelegate.foo()
- for each constructor of the delegate, create a similar constructor, which will initialize the myDelegate variable.

The transformation is implemented in a library and a script.

**How it works**

**The library: delegates.ijl**

```inject
library delegates {
  public void delegateInto(class delegating, class delegate) {

  string : paramList = ""; // Parameter list for each method
  string : paramPass = ""; // Passed parameters for the method
  string : paramSig = ""; // Parameter signature
  string : paramName;
  integer: globalCountVar;
  bool : hasOtherMethod;
  method : otherMethod;
  string : returntext;
  string : temp;

  in class delegate do {
    console("Delegate class is '<delegate.name>'");
    in class delegating do {
      add to members ${private <delegate.name> my<delegate.name>;}$
    }$
  }

  foreach method '(*)(*)' <=m> in delegate.methods do {
    if (m.isPublic) do {
      paramList = "";
      paramPass = "";
    }
    else paramSig = m.returnType.name+"+m.name";
  }

  // Constructors have different signature, than normal methods.
  if (m.isConstructor) do paramSig = delegating.name+"(";
      else paramSig = m.returnType.name+"+m.name+";

  The function operates on the public members of the delegate class. It will process the methods, and create the appropriate methods in the delegating class.
```

Page 30 of 44

The Inject/J Tutorial

Page 30 of 44
```javascript
hasOtherMethod = false;

console("Processing method '", m.signature, '");

// Now create a parameter listing
iterate(globalCountVar, m.parameters) {
    if (globalCountVar > 0) do {
        paramPass = paramPass + ", ",
        paramList = paramList + ", ",
        paramSig = paramSig + ", ";
    }

    paramName = m.parameters.getElement(globalCountVar).name;
    paramType = m.parameters.getElement(globalCountVar).staticType.name;
    paramPass = paramPass + paramName;
    paramList = paramList + paramType + " " + paramName;
    paramSig = paramSig + paramType;
}

paramSig = paramSig + ")";

The method signature is now complete.

iterate(globalCountVar, delegating.methods) {
    temp = delegating.methods.getElement(globalCountVar).signature;
    if (!delegating.methods.getElement(globalCountVar).isConstructor) do temp = delegating.methods.getElement(globalCountVar).returnType.name + "
    if (temp.equals(paramSig)) do {
        hasOtherMethod = true;
        otherMethod = delegating.methods.getElement(globalCountVar);
    }
}

Check, whether the generated signature is already implemented in the delegating class. If there is such a method, the
hasOtherMethod boolean variable is set to true, and the otherMethod variable is set to the name of the method.

if (hasOtherMethod) do console("There is a method with the same signature.");

if (!m.isConstructor) do {
    if (hasOtherMethod) do console("It is impossible to make the transformation! U

If the problematic method is not a constructor, we have to give up.

else {
    if (m.returnType.name.equals("void")) do returntext = ";
    else returntext = "return ";

If there is no problematic method, the script checks, whether the original method is void, and sets the return string
respectively.

in class delegating do {
    add to members ${
        public <m.returnType.name> <m.name>(<paramList>) {
            <returntext> my<delegate.name>.<m.name>(<paramPass>);
        }
    }
} $;

The function inserts the new method into the class definition.

} else {
    if (hasOtherMethod) do {
        in method otherMethod do {
            after $
```
If the delegating class already has a constructor with the same signature as the delegate class, the script adds the variable initialization code to the end of the constructor.

```plaintext
} else {
  in class delegating do {
    add to members ${
      public <delegating.name>({paramList}) {
        this();
        my<delegate.name> = new <delegate.name>({paramPass});
      }
    }$;
  }
}
```

Or it creates the new constructor.

```plaintext
}
}
}
```

Known limitation: the exceptions are not handled properly.

**The script: delegate.ij**

```plaintext
use library delegates;

class: delegated;
class: delegating;

script delegate {
  // Ask for classes, which need to be singletonized
  ask("Please select a class which will be a delegate.", namespace, delegated);
  ask("Please select a class which will be delegating.", namespace, delegating);

  delegates.delegateInto(delegating, delegated);
}
```

The script simply asks for the two classes to operate on, and calls the library function.

**Example Session**

To run this script, one needs to choose a `delegate` class and a `delegating` class. The output is the transformed delegating class.

**Chapter 7. Lesson 4: A complex example**
How to solve complex problems with Inject/J

A complex example: visualization of classes

Visualization is a very important aspect in almost every application. In this lesson we will create a simple visualization to a class, using Swing.

Our visualizing class is very simple: it displays a window with a table, which shows the actual values of the attributes of our class.

Figure 7.1. Data visualization window of a test class

On every visualized class the script performs the following operations:

- Modifies every constructor to include an array which contains the names of the attributes of the class. In addition the Visualizer instance is created with the attribute names passed as parameter.

- Creates a new method updateVisualization(), which collects the informations from the attributes of the class, and updates the Visualization using the Visualizer's update() method. It passes an array of stringified attribute values to the method.

We use the library mechanism of Inject/J to structure the code.

The datavis library

The whole library source can be found here: datavis.ijl

How it works

library datavislib {

Library declaration.

    // Implements get method for given attribute
    public void createGetMethod(class c, attribute a) {

Library function: implements get method for an attribute of a class.

    string : methodSignature;
    methodSignature = "get"+a.name+"()";

    in class c do {
        if not exists method methodSignature do {
            add to members ${
                public <a.staticType.name> <methodSignature> {
                    return <a.name>;
                }
            }$;
        }
    }
}
If the method doesn't exist, creates it.

```java
}
}

public void createSetMethod(class c, attribute a) {

Creates a set method for an attribute of a class.

```java
string : methodSignature;
methodSignature = "set"+a.name+"("+a.staticType.name+");"
in class c do {
    if not exists method methodSignature do {
        add to members ${
            public void set<a.name>(<a.staticType.name> <a.name>) {
                this.<a.name> = <a.name>;
            }
        }$;
    }
}

If the method doesn't exist, creates the method.

```java
}
}

public void createVisualization(class c) {

Create visualization for the class.

```java
string : className;
string : tableMembers;

This string holds the code for creating the array that holds the attribute names in the Java program.

```java
string : updatedMembers;

The string holds the code for updating the attributes in the Java program

```java
string : objectType;
attribute : a;
integer : tableCounter;
integer : accessCounter;
list(access) : accesses;
access : acc;

The following part generates the contents of the `tableMembers` and `updatedMembers` variable. The `tableMembers` variable contains Java code for instantiating a String array called "membernames". The array holds the names of the attributes of the class. It is passed to the constructor of the Visualizer class. The `updatedMembers` variable contains Java code for instantiating a String array called "updatemembers". The updatemembers array contains the string representation of the values set in the attributes of the class. It is passed to the `updateVisualization()` method.

```java
tableMembers="membernames=new String[<c.attributes.size>];\n";
updatedMembers="updatemembers=new String[<c.attributes.size>];\n";
updatedMembers=updatedMembers.resolveExpressions();
tableMembers=tableMembers.resolveExpressions();
```
The `resolveExpressions()` function of the string type evaluates the variable expansion expressions in the string.

```java
iterate (tableCounter, c.attributes) do {
    a = c.attributes.getElement(tableCounter);
    tableMembers = tableMembers + "membernames[<tableCounter>]=" + <a.name> + "\n"
    tableMembers = tableMembers.resolveExpressions();
}
```

Adding name of the current attribute to the membernames array.

```java
if (!a.hasPrimitiveType) do {
    updatedMembers = updatedMembers + "\n"
    updatedMembers = updatedMembers + "if (<a.name> == null) {
    updatedMembers = updatedMembers + "updatemembers[<tableCounter>] = " + <a.name> + "\n"
    updatedMembers = updatedMembers + "} else {
    updatedMembers = updatedMembers + "updatemembers[<tableCounter>] = <a.name>\n"
    }
    updatedMembers = updatedMembers.resolveExpressions();
}
```

If the attribute has no primitive type, it is possible to use the toString() method of the type. Of course, it needs to be checked, whether it is null.

```java
} else {
    objectType = datavislib.getObjectTypeForType(a.staticType.name);
    updatedMembers = updatedMembers + "updatemembers[<tableCounter>] = \n"
    Encapsulates the primitive value, and uses the object's toString() method.
    }
    updatedMembers = updatedMembers.resolveExpressions();
```

If the attribute has primitive type, it uses a library function, to get the corresponding object type.

```java
in class c do {
    add to imports ${ visualizer.Visualizer }$;
    add to members ${
        private Visualizer _visualizer = null;
    }
    
```

Transforms the class.

```java
private void updateVisualization() {
    String[] updatedmembers;
    _visualizer.update(updatedmembers);
}
```

Adds a private attribute to the Visualizer class. The Visualizer class needs to be part of the sources. It has to implement one constructor with a string array parameter, that holds the attribute names.

The visualizer class implements another method `update()` with a string array parameter. It holds the updated values of the attributes. The `updateVisualization()` calls this method.
foreach method '*(*)' <=m> in c.constructors do {
  in method m do {
    after ${
      if (this._visualizer == null) {
        String[] membernames;
        <tableMembers>
        this._visualizer = new Visualizer(membernames);
        this.updateVisualization();
        this._visualizer.show();
      }
    }$;
  }
}

Code will be inserted in every constructor to create the Visualizer instance.

foreach attribute '*' <=attr> in c.attributes do {
  if (!attr.name.equals("_visualizer")) do {
    accesses = attr.referencingAccesses;
    iterate (accessCounter, accesses) do {
      acc = accesses.getElement(accessCounter);
      if (acc.isWriteAccess) do {
        in access acc do {
          after ${
            this.updateVisualization();
          }$;
        }
      }
    }
  }
}

After each write access call to the updateVisualization() method is inserted.

public string getObjectTypeForType(string typeName) {
  string : result;
  if (typeName.equals("boolean")) do {
    result = "java.lang.Boolean";
  }
  else if (typeName.equals("char")) do {
    result = "java.lang.Character";
  }
  else if (typeName.equals("byte")) do {
    result = "java.lang.Byte";
  }
  else if (typeName.equals("short")) do {
    result = "java.lang.Short";
  }
  return result;
}
Data Visualizer

use library datavislib;

string : getMethodSignature, setMethodSignature;

script datavisualizer {
    foreach class '::*' <=c> in namespace do {
        if (!c.simpleName.equals("Visualizer") & c.attributes.size>0) do {
            datavislib.createVisualization(c);
        }
    }
}

For each class in the namespace creates the visualization.

Example Session

To run this script, one needs to include a class to transform, and the Visualizer class in the namespace. The output is the visualized class.

Chapter 8. Lesson 5: Adaptive Programming using Inject/J

Table of Contents

- Introduction to Adaptive Programming
  - The AP-based model of Inject/J
  - Language elements of the AP extension
- Example 1: Hello World in Adaptive Programming
  - How it works
  - Example Session

After completing this lesson you will understand:

- The basics of adaptive programming.
- How to use Inject/J to write adaptive programs.

Introduction to Adaptive Programming
The complex software systems used nowadays bring up many new problems in the area of maintainability, customization and stability. Many new methods are being developed to deal with this kind of problems, and one of these is Adaptive Programming (AP). The key benefits of AP are:

- Due to the separation of structure and functionality, it is possible to define new methods independently of program source.
- Specification at a more abstract level instead of the implementation level.
- Not all participant classes have to be explicitly named.
- Scalability: it can be applied to small and large systems as well.
- The developer does not have to worry about the necessary method calls. They will be automatically computed, and injected into the source code.

The AP-based model of Inject/J

Instead of directly modifying the source code, the AP developer writes a metaprogram, that consists of the following parts.

- Signature of the new method
- Specification of the participant classes
- A set of source code injections

The AP model of a program is a directed graph. The nodes (vertexes) of the graph are the classes of the program, the edges are the associations between them. There are different kinds of vertexes and edges depending on the program structure.

The key benefit of AP is that the developer only needs to find the participant analysis classes. The participant implementation classes are automatically computed and modified. Of course this brings up some concerns. The developer does not fully specify the participant classes, and it is possible that he thinks something different about the system structure and does not know the actual structure. There should be tools available for the graphical visualization of the model graph. Currently Inject/J does not provide such a tool. There is a system named GOOSE developed by the FZI which is able to visualize software system structures.

The original AP model makes very strict assumptions of the system structure. To construct a tool applicable to existing systems, some rules have to be broken, and a new model have to be built. In the following sections, the elements of the new model will be explained.

Node types

Interface nodes

Every interface class is represented by an interface node. An interface class cannot contain method bodys.

Navigation nodes

A class that can contain method bodies is represented by a navigation node.

Repetition nodes

A class that can contain other object references is represented by a repetition node. A typical example is the java.util.Vector class from the Java Collection Framework. The classes represented by repetition nodes are called repetition classes. In most cases repetition classes are implementation classes, therefore they do not need to be selected by the developer. The repetition nodes are special navigation nodes. An array can be represented by a repetition node.

Alternative nodes
Alternative nodes are special navigation nodes. A class is represented by a navigation node, if one or more class inherits from it.

**Edge types**

There are 3 different kinds of associations between nodes.

- **Attribute association:** A class named "A" has an attribute of type class "B" named "b". A method call can be forwarded by calling a method of attribute "b".

- **Inheritance association:** the method call forwarding from a base class to a subclass is implicit, as the subclass inherits the methods from the base class. Delegation from the subclass to the base class can be done by using an explicit method call.

- **Interface implementation:** The delegation path goes from an interface to a class that implements the interface.

Edges represent paths to forward method calls. There are 5 different types of edges.

**Attribute edge**

An edge with a navigation or interface node as target.

**Repetition edge**

An edge with a repetition node as target.

**Alternative edge**

If both ends of an inheritance association from a base class to a subclass are navigation nodes, the association is modelled using an alternative edge.

**Interface edge**

If both ends of an inheritance association from a base class to a subclass are interface nodes, the association is modelled using an interface edge.

**Inheritance edge**

Inheritance association from a subclass to a base class is modelled using inheritance edge, if both classes are navigation nodes.

**Language elements of the AP extension**

**Propagation pattern**

The propagation pattern is the central element of the AP extension. It can be described as follows:

```xml
<PropagationPattern> ::= <NavigationStatement> <MethodDescription> '{'
    <PathFunctionality> '}'
```

The navigation statement describes the paths to operate on. The method description gives the signature of the new operation. The path functionality describes the functional code fragments to be injected to the specified paths.

**Navigation Statement**

The navigation statement describes the paths to operate on. One can give the path by describing the nodes and the edges.

```xml
<NavigationStatement> ::= 'in' <NodeSelection> 'do'
    'add' 'navigation'
```
Specifying starting nodes

In order to specify a path, the starting points have to be defined.

in <NodeSelection> do

The statement defines starting points. The NodeSelection statement is a comma separated list of nodes. (e.g. A,B,C) The special value * matches every node in the model.

Most of the time the set of starting nodes contains exactly one element. Specifying more than one starting node is a good idea, when the target nodes and the functionality to inject are the same for every class.

The target node specification is optional. If omitted, it has the same effect as 'to '*' . The keyword to_stop changes the behaviour to end every path in the specified target nodes. This way the edges starting in the target nodes are not considered.

Specifying waypoints on the paths

Using the via and the not via keywords in conjunction with a list of nodes one can specify nodes to visit or not to visit. This approach is useful when there are more possible paths from the starting node to the end node.

Disallowing cycles

If the noCycles keyword is used, no cycles will be allowed on the specified Delegation path, only direct cyclefree paths from the start to the target nodes. The edges that point back on a node on the path will be ignored, and so no cycle is possible.

Edge Selection

<EdgeSelection> ::= (( 'Attribute>' | 'Repetition>')

'(' <NodeSelection> ',,' <Name> ',,' <NodeSelection> ')') |

(( 'Alternative>' | 'Inheritance>' | 'Interface>') '('

<NodeSelection> ',' <NodeSelection> ')'

)

The edge selection construct selects an edge from the path set. One can set the edge type to be selected (Attribute, Repetition, Alternative, Inheritance, Interface) and the nodes found on the ends of the edge. For the first Attribute and Repetition edges one has to specify the name of the attribute in the class used for the implementation of the edge.

Method description

<MethodDescription> ::= 'add' 'method' <Visibility> <ReturnType> <MethodName>

'(' <ParameterList> ')' (<Exceptions>)? (<Comments>)?

Declares the method signature to be inserted on the selected path set. A method will be injected for each node in the path. The following example shows the method injected in a starting Node of an attribute edge.

<Visibility> <ReturnType> <MethodName>(...) [throws <Exceptions>] {

// Navigation statement to the next node
attrib1.<MethodName> {...};

...
Source code injection

In this section

\[
\langle \text{SourceInjection} \rangle ::= \text{'in'} (\langle \text{NodeSelection} \rangle | \langle \text{EdgeSelection} \rangle ) \text{'do'}
\]
\[
\langle \text{beforeNavigation} \rangle | \langle \text{afterNavigation} \rangle
\]
\[
\langle \text{'${'} \langle \text{SourceCode} \rangle \text{'}$} \rangle
\]

The source injection statement inserts a specified source code fragment to the selected path nodes.

If the NodeSelection is used the injected method in the selected nodes are supplemented by the specified code fragment. In case of beforeNavigation the code is inserted before the navigation statements, in case of afterNavigation the code is inserted after the navigation statements.

If the EdgeSelection is used the semantics is different. In case of beforeNavigation the code fragment is inserted in the starting node of the selected edges before the navigation statements. In case of afterNavigation the fragment is inserted in the target node of the selected edges.

Path global variables

\[
\langle \text{PathGlobalVariable} \rangle ::= \langle \text{Type} \rangle \langle \text{Name} \rangle
\]

It is possible to specify path global variables. These variables are accessible in every method on the path. Every change in the variable's value is propagated in both directions: to the caller method and the called methods. If a more strict change propagation semantics is desired the programmer has to ensure it in the source code injection blocks.

Path aliases

\[
\langle \text{APAliasStatement} \rangle ::= \text{'path'} \langle \text{Name} \rangle
\]
\[
\langle \text{NavigationStatement} \rangle
\]

It is possible to define an alias to a navigation statement. As described earlier this statement specifies a path set.

\[
\langle \text{APCallStatement} \rangle ::= \text{'traverse'} \langle \text{Name} \rangle \text{'('}
\]
\[
\langle \text{MethodDescription} \rangle \text{'{'}
\]
\[
\langle \text{PathFunctionality} \rangle \text{''}\text{')'}
\]

On this named path set one can perform arbitrary code transformations.

Example 1: Hello World in Adaptive Programming

The first example deals with a simple model of a car. There are 4 classes in the model: Car, Chassis, Axle and Wheel. The UML class diagram is shown on the next figure.

Figure 8.1. UML class diagram for the Car model
The task is to print out the tire pressure for every wheel in a car. The following script adapts the program to solve this task.

```inject
script aphello {
  in class 'Car' do {
    add navigation {
      to 'Wheel'
      add method 'void printTirePressure()' {
        in 'Wheel' do
          beforeNavigation ${
            System.out.println("Pressure of the tire:"+tirePressure)
          }$
      }
    }
  }
}
```

**How it works**

After declaring the script header, the script navigates to the starting node 'Car'. It sets up a navigation path from the 'Car' node to the 'Wheel' node.

```inject
add method 'void printTirePressure()' {
  in 'Wheel' do
    beforeNavigation ${
      System.out.println("Pressure of the tire:"+tirePressure)
    }$
}
```

It needs to add a new method with the given signature on this path.
It adds the actual functionality to the method in the Wheel node.

Using this approach it is possible to separate the functionality from the structure. Many problems can be solved by using such delegation patterns and the adaptive programming features of Inject/J.

**Example Session**

The original sources are: Car.java Chassis.java Axle.java Wheel.java

The transformed sources are: Car.java Chassis.java Axle.java Wheel.java

[3] A metaprogram is a program, that modifies other programs or itself. In this respect a compiler is a metaprogram.

**Chapter 9. Feature Matrix of the Tutorial**

In this section we list the features of Inject/J along with the reference to its explanation.

**Table 9.1. Feature matrix of Inject/J**

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Feature explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace concept</td>
<td>the section called “example 1: The visitor”</td>
</tr>
<tr>
<td>Navigation statements</td>
<td></td>
</tr>
<tr>
<td>Hierarchical navigation</td>
<td>the section called “example 1: The visitor”</td>
</tr>
<tr>
<td>Direct navigation</td>
<td>the section called “example 2: Visitor with direct navigation”</td>
</tr>
<tr>
<td>The embed statement</td>
<td>the section called “example 2: The interface extractor”</td>
</tr>
<tr>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>Global variables</td>
<td>the section called “example 2: Visitor with direct navigation”</td>
</tr>
<tr>
<td>Local variables</td>
<td>the section called “example 1: The visitor”</td>
</tr>
<tr>
<td>Expressions</td>
<td></td>
</tr>
<tr>
<td>Variable usage expressions</td>
<td>the section called “example 1: The comparable interface”</td>
</tr>
<tr>
<td>Conditional expressions</td>
<td></td>
</tr>
<tr>
<td>Assignment expressions</td>
<td></td>
</tr>
<tr>
<td>Script declarations</td>
<td>the section called “example 1: The visitor”</td>
</tr>
<tr>
<td>Weave statements</td>
<td></td>
</tr>
<tr>
<td>before</td>
<td>the section called “example 3: A method call trace”</td>
</tr>
<tr>
<td>after</td>
<td>the section called “example 3: Delegation”</td>
</tr>
<tr>
<td>replace</td>
<td></td>
</tr>
<tr>
<td>change modifier</td>
<td>the section called “example 1: The singleton design pattern”</td>
</tr>
<tr>
<td>add to members</td>
<td>the section called “example 1: The comparable interface”</td>
</tr>
<tr>
<td>add class</td>
<td></td>
</tr>
<tr>
<td>add interface</td>
<td>the section called “example 2: The interface extractor”</td>
</tr>
<tr>
<td>add to imports</td>
<td>the section called “ A complex example: visualization of classes ”</td>
</tr>
<tr>
<td>add to extends</td>
<td></td>
</tr>
<tr>
<td>add to implements</td>
<td>the section called “ example 1: The comparable interface ”</td>
</tr>
<tr>
<td>add to throws</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>the section called “ example 1: The visitor ”</td>
</tr>
<tr>
<td><strong>Conditional statements</strong></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>the section called “ example 1: The comparable interface ”</td>
</tr>
<tr>
<td>if in</td>
<td></td>
</tr>
<tr>
<td>if exists</td>
<td></td>
</tr>
<tr>
<td><strong>User interaction statements</strong></td>
<td></td>
</tr>
<tr>
<td>console</td>
<td>the section called “ example 1: The visitor ”</td>
</tr>
<tr>
<td>message</td>
<td>the section called “ example 1: The comparable interface ”</td>
</tr>
<tr>
<td>ask</td>
<td>the section called “ example 2: Visitor with direct navigation ”</td>
</tr>
<tr>
<td><strong>Other statements</strong></td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>the section called “ example 1: The comparable interface ”</td>
</tr>
<tr>
<td>iterate</td>
<td>the section called “ example 2: The interface extractor ”</td>
</tr>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>the section called “ example 1: The singleton design pattern ”</td>
</tr>
<tr>
<td><strong>Weavepoint types</strong></td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>the section called “ example 1: The visitor ”</td>
</tr>
<tr>
<td>class</td>
<td>the section called “ example 1: The visitor ”</td>
</tr>
<tr>
<td>attribute</td>
<td>the section called “ example 1: The comparable interface ”</td>
</tr>
<tr>
<td>access</td>
<td>the section called “ A complex example: visualization of classes ”</td>
</tr>
<tr>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>the section called “ example 2: Visitor with direct navigation ”</td>
</tr>
<tr>
<td>return</td>
<td>the section called “ example 1: The visitor ”</td>
</tr>
<tr>
<td>exception</td>
<td></td>
</tr>
<tr>
<td>exception throw</td>
<td></td>
</tr>
<tr>
<td>local variable declaration</td>
<td></td>
</tr>
<tr>
<td>explicit</td>
<td></td>
</tr>
<tr>
<td>parameter</td>
<td>the section called “ example 2: The interface extractor ”</td>
</tr>
<tr>
<td><strong>Variable types</strong></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>the section called “ example 2: The interface extractor ”</td>
</tr>
<tr>
<td>integer</td>
<td>the section called “ example 2: The interface extractor ”</td>
</tr>
<tr>
<td><strong>Adaptive Programming features</strong></td>
<td></td>
</tr>
<tr>
<td>AP example</td>
<td>the section called “ Example 1: Hello World in Adaptive Programming ”</td>
</tr>
</tbody>
</table>