MOVE | Smalltalk to Java

Productoverview

Document Version 1.2

GEBIT Solutions

I. MOVE/Smalltalk to Java - Introduction

Abstract

In the currently rapidly changing world of business computing many long-term IT-projects are out of date before they are deployed. The reasons for this failure are constant changes in business processes along with constantly changing development tools. While short ago the programming languages of choice were C++ and Smalltalk and client-server-computing was state of the art, today Java and network computing are the buzzwords that symbolize these changes.

In practice you find many of these issues in Smalltalk applications. Java is established as a new technology offering features like

- Ability to run over the Internet - Java applications can be run in a WEB-Browser.
- Automatic Software Distribution - this features promises to take away some of the headache caused by expensive maintenance of client seats.
- Thin Clients Architecture - this architecture allows to run applications over low speed connection lines efficiently and brings your business services to the client directly.

It is very uncertain that these features will be implemented by the remaining Smalltalk vendors. But even if so, to be buzz and compatible and not only doing a „marketing hack“, somebody has to take a very close look. So there are reasons to move to the Java platform. On the other hand existing Smalltalk applications and business logic implements an important part of your business today.

"MOVE/Smalltalk to Java" (MOVE/S2J in its more convenient spelling) developed by GEBIT helps you migrating parts of your ObjectStudio Smalltalk applications to Java. Theoretically any Smalltalk class can be converted and the process of conversion can be configured. MOVE/Smalltalk to Java offers an important tool-set to facilitate this task not trivial. Although you will be never able to automatically convert 100% of your code, this tool-set takes away a big chunk of that work from you and helps you to save considerable costs.

What is MOVE/Smalltalk to Java?

MOVE/Smalltalk to Java is a framework, developed by GEBIT, that helps to convert applications written in *ObjectStudio* Smalltalk to Java. MOVE/Smalltalk to Java is written in Smalltalk itself and will be shipped with the complete source code. The classes, that make up MOVE/Smalltalk to Java are designed, so that the can easily be customized. Currently only the ObjectStudio dialect is supported.

In the following chapters we will talk about general source code migration aspects as well as we give an overview over the components and the principal workflow with MOVE/Smalltalk to Java.

Source Code Migration in General

Tool-Based Migration Expectations

A migration from one language to another is useful only, if a specific set of requirements is met. The following points summarize the various expectations considering a tool-based migration to a different platform. MOVE/Smalltalk to Java tries to meet this set of expectations. How? See below!

Reduced migration costs

Quite obvious is, that a tool-based migration should in particular reduce the migration costs. How much is saved, depends on a variety of factors. A first approach will try to measure, how much percent of the source code can be converted automatically. The 1st difficulty in this approach is, that the units for measuring such a percentage are not easy to define. One could count "lines of code", that can be converted without manually interference. The biggest disadvantage of that approach is, that converting Smalltalk to Java will not necessarily result in a linear amount of code. One Smalltalk class will be converted best into five Java types, while another will convert to a single Java class. A better approach is to count the development time, that is saved. The saved time is not easy to measure. We recommend parallel feasibility tests converting single comparable classes manually and using a tool like MOVE/Smalltalk to Java.

Improvement in architecture

A pure conversion of the classes of your project is not very attractive. The only advantage of such a conversion is to escape from the dependency of your Smalltalk solutions provider. If you want to benefit from the features modern Java applications offer, you should rearchitect your software on the way.

Integration with the target platform

The generated Java classes should conform to the Java platform. They should build on top of standard Java packages, products and APIs. Otherwise your project will not be maintainable. It will still remain a legacy application.

Maintainability

The generated source code should be maintainable. It should conform to standard Java programming style guides and design patterns rather than to Smalltalk paradigms.

Natural or easy-to-use migration tool

Nobody using a tool to migrate software is willing to invest a lot of money to learn the tool. Most likely the tool should be seamlessly integrated into the "known" development process on either the source or destination platform.

Replacement process of existing application

Usually you will be migrating a mission critical application, that is in production use. Even a tool-based migration will take some time. During this time, you will see change requests onto the existing legacy application, that have to be rolled in. Of course, those changes have to make it also into the migrated Java application. A good version tracking concept or even a redo the conversion of some of the source code is required, that helps you to keep track up with those changes.

When can I use a Tool for a Migration?

If you feel uncertain, about whether you should be migrating a Smalltalk application at all or whether you should be using a tool to do so, answering the following questions will help you to come to a decision.

Why do you want to migrate?

The more reasons of the following questions apply, the more it is recommended for you to change the platform.

- Are you unsatisfied with the vendor of your Smalltalk tool?
- Do want to benefit from new technologies like the WEB-enablement?
- Do you have serious problems with the client administration? Is the rollout of new versions complicated or expensive?
- Do you have problems finding employees willing to be educated and developed in Smalltalk?

How is the quality of the existing source code?

The better the quality of the existing source is, that higher are the benefits from using a tool for a migration. To measure the quality, try to answer the following questions.

- Is the architecture of your application divided into logical tiers?
- Was a style guide enforced during the development of the Smalltalk application?
- Does the existing application build on a framework, that enforces uniform programming and design patterns?

What is the knowledge of your development team?

No tool works without support by good people. Moving an application from Smalltalk to Java will not work, if the people performing that move do not have skills in both programming languages.

- Does the development staff have Smalltalk skills?
- Does the development staff have good Java skills?
- Does the development staff understand client-server architecture principles very well?

How big is the codebase to be converted?

The bigger your project is, the better the percentage of savings of a tool-based migration is. Using a tool implies on hand a certain trade-off in training. On the other hand, the customization of a tool like MOVE/Smalltalk to Java raises the efficiency of the tool.

For us an automatic conversion starts making sense where about 80-100 Smalltalk classes are involved. This is a rule of thumb as a good class is usually a small and lean class in which case this number may actually be a little bit higher.

Different Migration Scenario

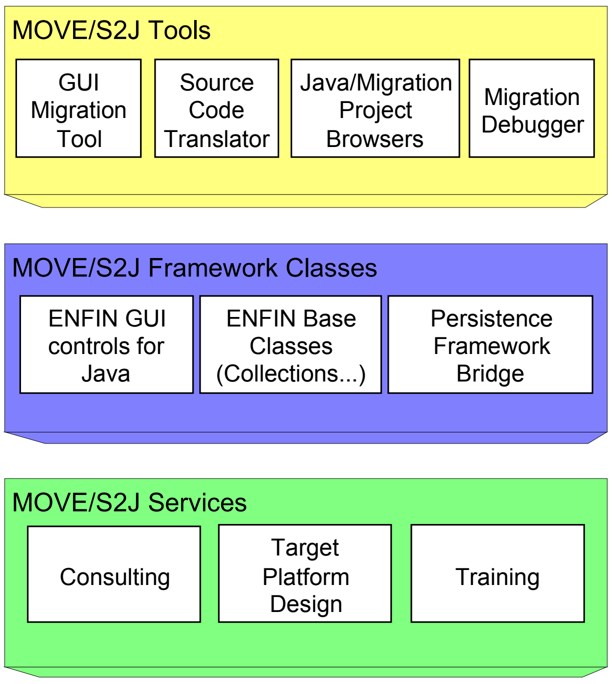
There are several reasons and organizational constraints that can effect the model, in which you can migrate a Smalltalk application to the Java programming language.

- Scenario 1: Java GUI front-end to Smalltalk back-end
  If you are primarily interested in enabling your application for the web, you may decide to concentrate on creating a thin-client Java front-end to your Smalltalk application. MOVE/Smalltalk to Java will support you in migrating the GUI and the business objects and will connect the Smalltalk server objects via CINCOMS product DCF with the thin client GUI.

- Scenario 2: Complete Migration to Java 2-Tier Application
  If you want to benefit from Javas platform independency on the server side (availability on OS390 for instance), you might decide to fully convert your Smalltalk project.
- Scenario 3: Complete Migration to Java 3-Tier Application
  If you want to run distributed, you might be interested in changing the architecture of your application by moving from a physical 2-tier Application (fat client talking to the Database directly) to a 3-Tier application server.
- Scenario 4: Migration of particular libraries / vertical frameworks
  If you have developed a Smalltalk business object library, that should be used in new Java applications, you might as well simply migrate those objects.

Components

MOVE/Smalltalk to Java is not sold as an off the shelf-product, but in combination with consulting services from GEBIT. It consists of the following components.


MOVE/S2J Tools
GUI Migration Tool
Source Code Translator
Java/Migration Project Browsers
Migration Debugger
MOVE/S2J Framework Classes
ENFIN GUI controls for Java
ENFIN Base Classes (Collections...)
Persistence Framework Bridge
MOVE/S2J Services
Consulting
Target Platform Design
Training

The MOVE/S2J Tools. The MOVE/S2J project browser helps you to keep track of the Java classes you have already created. It also helps you testing the generation process by providing hooks to compiling and running the Java application. You can also start the migration wizard, that converts ENFIN controller / interfacecomponent classes from here. Other Tools help you to tune the conversion process.

The MOVE/S2J Source Code Translator. The user interface to this functionality is the Convert Class wizard dialog, that can be started from the MOVE/S2J project browser. You may use the source code translator programmatically as well. It is written in Smalltalk and can be customized.

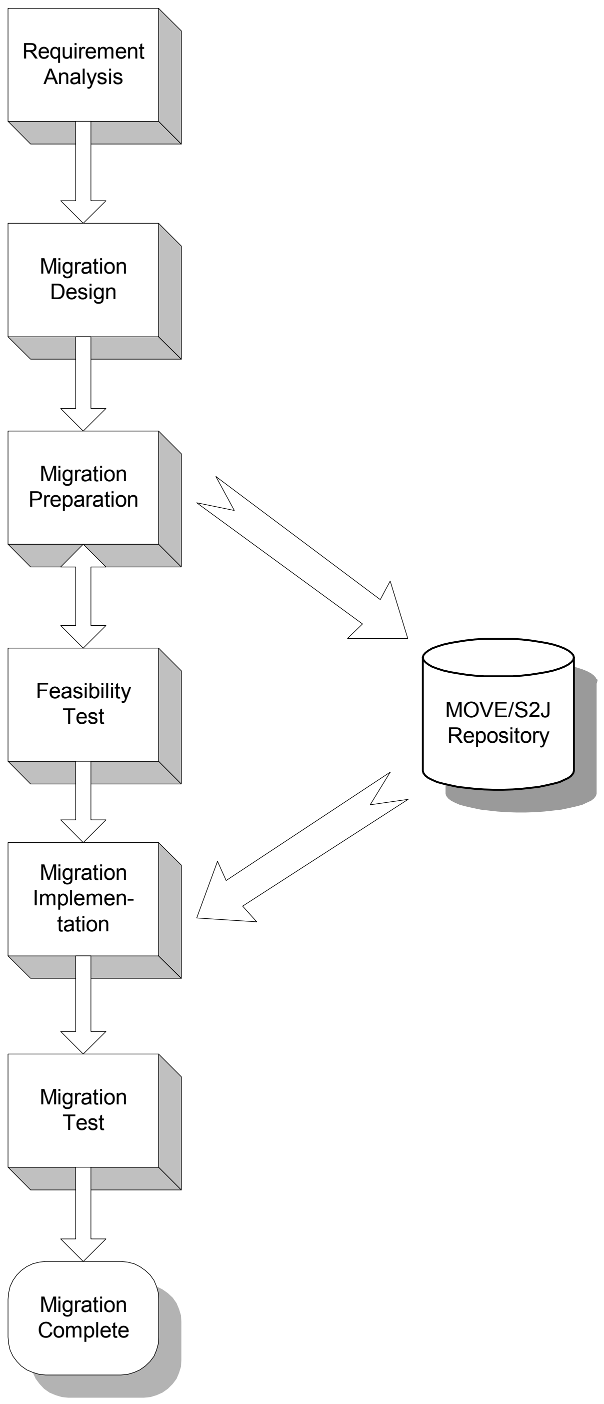
The MOVE/S2J Java Framework classes. These classes are located in the ZIP-file S2J.ZIP. The documentation to those classes is available from the doc directory and can be browsed with every web browser. Install shield adds this file to your CLASSPATH environment variable.

Some helper tools, if you are using CINCOMs product DCF to create a distributed application with a Smalltalk application server and Java Applets/Applications to talk to the Smalltalk application server.

Documentation of the ENFIN Java framework classes.

Development Workflow

An MOVE/Smalltalk to Java migration project can be planned with the following major steps.



1. Requirement Analysis*:* Define your migration scope. To do so, analyze your reasons to move to Java and your concrete project requirements. The result of this step is the decision for one of the scenarios given above.

2. Migration Design*:* Identify the programming patterns used in your Smalltalk application. At the same time, define your style guide and the design patterns you want to use in the Java application. If in doubt, consult the GEBIT Java Framework (TREND) recommendations. Try to define how programming patterns from the Smalltalk application can be mapped to Java.
Example Pattern: Your Smalltalk applications might test for errors by checking return codes explicitly. In Java you might want to use exceptions for this purpose.

3. Migration Preparation*:* Extend MOVE/S2Js repository containing the resource hints. To do so, use the MOVE/S2J tools like the MOVE/S2J event tracing facility and the MOVE/S2J Conversion Debugger. Also define conversion rules, that map the pattern identified in step 2.

4. Feasibility Test*:* Try to test convert some of the target classes for the migration process using the hints defined in step 3. If the results are not satisfying yet, go back to step 3.

5. Migration Implementation*:* Define the full set of classes to be migrated. Start migrating those classes using the MOVE/S2J project browser by converting the classes automatically, fixing migration problems manually. The MOVE/S2J project browser will provide a convenient environment to do so. Write little Java test scenarios to test the classes converted so far. If test classes existed implemented in Smalltalk, you might as well convert them and use them for testing.
Tip: Investing some time on a standard test procedure frame to be run as an application or an Applet might be considerable, in particular if the number of classes to be tested is big.

6. Migration Test*:* If enough classes are converted, you can start assembling the final application to test it.

The migration implementation step is however dependant on the migration scope as outlined before. In particular the migration implementation step, that involves the work with the MOVE/S2J project browser varies from scenario to scenario. In any case, the efficient use of the MOVE/S2J project browser is central to the migration implementation. Currently, server side definition of CORBA interfaces has to be performed currently manually. Tool support for this activity will be supported later.

II. MOVE/Smalltalk to Java - Tools

Cross compilation is only one of the challenges during a migration project. This chapter describes the MOVE tools, that guide you through and support the whole migration process.

- The MOVE/S2J project browser
- The MOVE/S2J migration task list
- The MOVE/S2J Repository Browser
- The MOVE/S2J Method Trace Tool The MOVE/S2J Project Browser

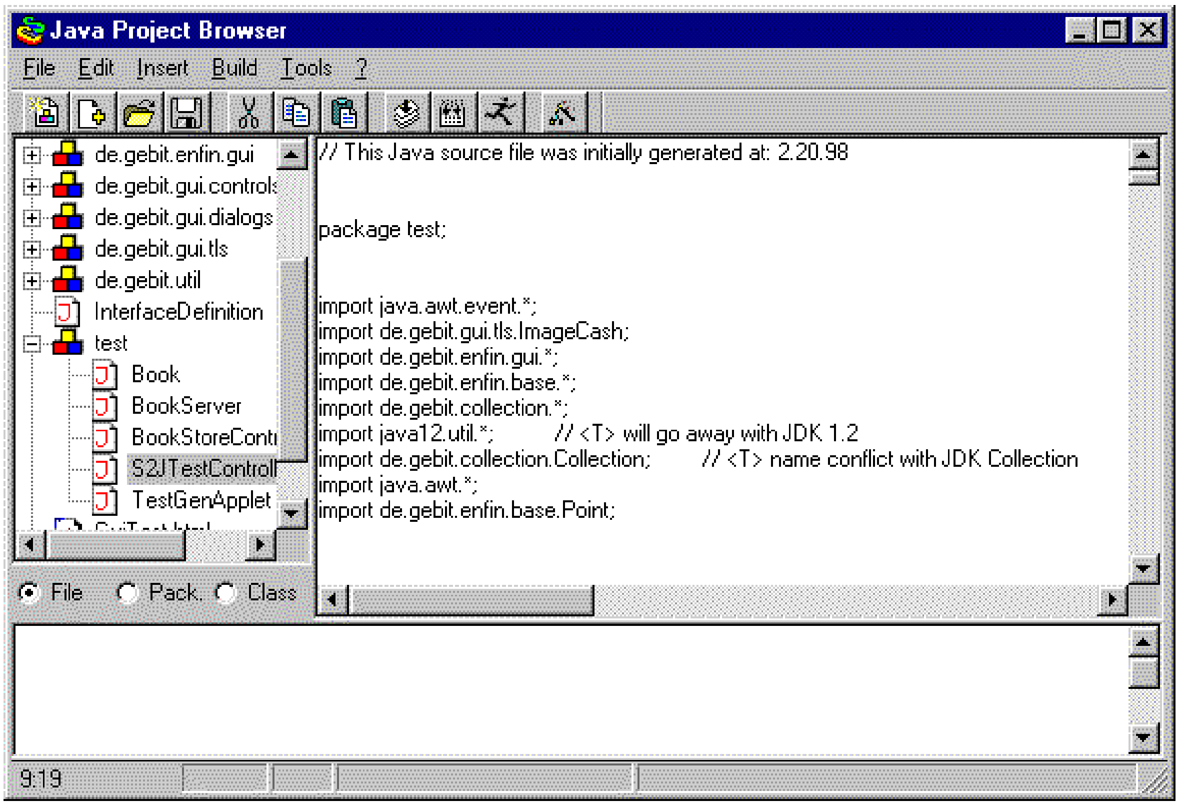
Purpose

The Project Browser helps you to keep track of the Classes / GUI Classes you have converted so far. You can edit the resulting Java source files from here and test the conversion results by compiling the classes and running the Java application in the applet viewer or a web browser.

User Interface

The MOVE/S2J Project Browser displays the following elements:

Java Package Tree Source Window



```
// This Java source file was initially generated at: 2.20.98


package test;


import java.awt.event.*;
import de.gebit.gui.tls.ImageCash;
import de.gebit.enfin.gui.*;
import de.gebit.enfin.base.*;
import de.gebit.collection.*;
import java12.util.*;          // <T> will go away with JDK 1.2
import de.gebit.collection.Collection;       // <T> name conflict with JDK Collection
import java.awt.*;
import de.gebit.enfin.base.Point;
```

Message Window

The Java Package Tree Window

displays the java source files sorted by files / packages or classes. Selecting a Java class here displays the associated Java source file in the source editor pane. Double clicking on an entry here will open the ObjectStudio text editor here to edit the source file.
This window displays the following symbols:

A Java package.
A folder.
A Java source file. This can be either a Java file created by migrating an object studio source file as also a handwritten Java file.
An HTML-file. Every java project can have one associated HTML-file, that contains the starter Java applet class reference for running your Java application.
Any other file such as documentation

The Message Window

The message window displays the results of the compilation of a Java source file. If the result contains errors or warnings, double clicking on the lone that contains the error will select the source file where the error occurred and highlight the line containing the error.

The Source Editor Window

The source editor window displays the source to be edited. The editor works in compliance with all other object studio editor windows. In addition pressing Ctrl+F1 on a word tries to find the associated Java help for the class / method name that is represented by the keyword. If this can be found, the Web-Browser is opened to display the associated help HTML file.

Working with the MOVE/S2J Project Browser

The following section describes how to work with the Java Project Browser.

Converting Classes and User Interfaces

You can convert smalltalk Classes, Controllers and InterfaceComponents by selecting *"Convert smalltalk to Java..."* from either the *Tools* menu or from the toolbar.
The Convert Class Wizard will assist you in converting a Smalltalk class / user interface by guiding you through the following steps:

- Select what you want to convert: a Controller, an InterfaceComponent or a Class
- Select the destination Java package, where the result should be placed
- If converting a Controller you may optionally
  - generate an HTML file that references the startup applet
  - generate a main applet / application class that starts up your controller
- Pressing the Finish button will

Insert Files into the project

With the menu *Insert->Files into project...* or by pressing the corresponding toolbar button, you may add additional files to the Java project. Depending on the type of the file you are inserting, it will be added to:
.java - Files
To the Java source file packages.
.html - File
This will be added to the files pane if it does not exist already. It also will be made the default (starter) html-file for your test application.
.*
You can add other files such as documentation to the project. Any other file will be added simply to the folder named "Other Files".

Creating new Java classes

With the menu *Insert->New Class...* or by pressing the corresponding toolbar button, you can create a new Java class to be inserted into your project. In the dialog, that opens, you can create a Java class with a default header. This dialog also assist you in creating the necessary directories and files. The files are created corresponding to Java file name conventions in the directory corresponding to the package name with the '.' in the package name replaced with \ characters. All files are created relative to the base directory of the project, that is determined by the method *S2JEnvironment>>javaDefaultProjectDir.*

Starting the compiler

With the menu *Build->Compile* or by pressing the corresponding toolbar button, you can start the Java compiler to compile the current selected Java source file. The actual Java compiler that is started depends on the method *S2JEnvironment>>javaCompiler*. When the compiler detects errors, the errors are reported into the message window. Double clicking on an error in the message window will bring you to the place where the error was found.

Testing your application

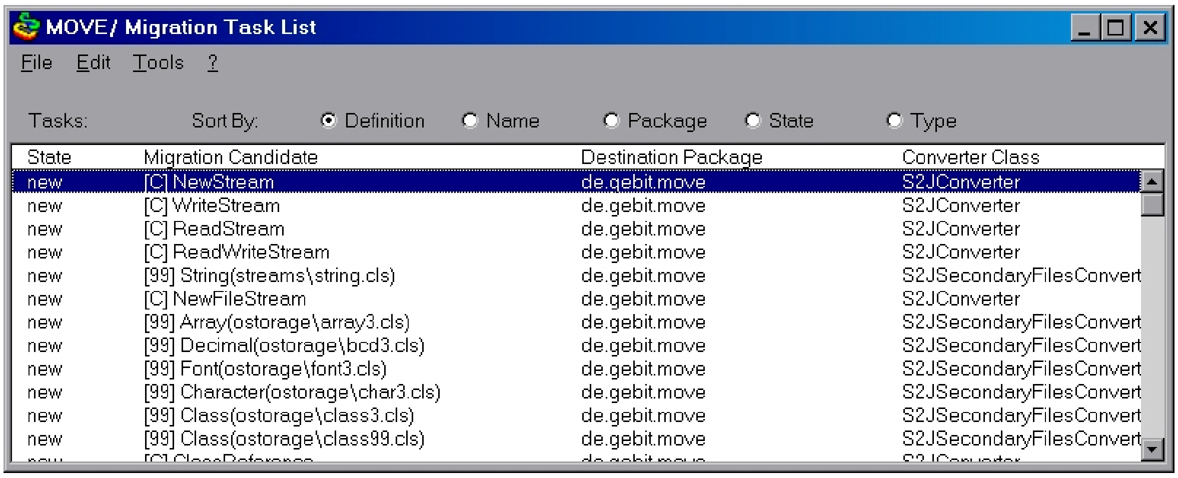
With the menu *Build->Run...* or by pressing the corresponding toolbar button, you can run the Java application from ObjectStudio. Which Java VM (JDK virtual machine / which WEB browser) actually is used to run the application is determined by the method *S2JEnvironment>>javaAppletViewer.*

The MOVE/S2J Migration Task List

Purpose

The migration task allows you to collect all objects (files, classes, ...) of a project to be migrated and cross-compile all project classes in a single step. It also allows for generating statistics information and for conveniently selecting classes to be migrated. The set of classes defined are maintained in a **migration project** which can be saved in later migration sessions. For every task defined, a state is maintained, which allows you to keep track of your migration project. When building task lists, it can automatically determine dependencies between tasks.

User Interface



Tasks: Sort By: [✓ Definition] Name Package State Type

State	Migration Candidate	Destination Package	Converter Class
new	[C] NewStream	de.gebit.move	S2JConverter
new	[C] WriteStream	de.gebit.move	S2JConverter
new	[C] ReadStream	de.gebit.move	S2JConverter
new	[C] ReadWriteStream	de.gebit.move	S2JConverter
new	[99] String(streams\string.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[C] NewFileStream	de.gebit.move	S2JConverter
new	[99] Array(ostorage\array3.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[99] Decimal(ostorage\bcd3.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[99] Font(ostorage\font3.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[99] Character(ostorage\char3.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[99] Class(ostorage\class3.cls)	de.gebit.move	S2JSecondaryFilesConvert
new	[99] Class(ostorage\class99.cls)	de.gebit.move	S2JSecondaryFilesConvert

The **task list** displays every object to be migrated, the destination package selected for the output file and the responsible handler for the object to be migrated. The task state shows, whether the migration is due, done or tested. The **Sort by** buttons allow you to change the order of the task list entries to be able to quicker find things already there or not.

Using the Migration Task List

Typically you will create a migration task list by simply opening the task list or by explicitly selecting

File>>New.

You start then adding files using one of the following *Edit* menu entries:

Edit>>Insert Class...

This function brings up a dialog, where you may select a class to be added. The migration task list will at this point try to automatically determine all dependant classes (and other items like pool dictionaries) and add them automatically in the right order. The class is always inserted, nevertheless, whether it is a "wellknown" converted class.

Edit>>Insert Subclasses of...

Allows to add a whole class hierarchy in the same way as described under *Add Class*. Only classes are inserted, which are not to be converted or a considered system classes.

Edit>>Insert Application...

This function allows you to add all classes contained in an ObjectStudio LoadableApplication text file. Dependant objects are not determined in this step. Only classes are inserted, which are not to be converted or a considered system classes.

Note: the spelling of the file names in the text file must match the spelling of the `class sourceDict` entries in ObjectStudio. If you installed your own ApplicationLoader facility, this might not be the case.

Edit>>Insert Application with dependants...

Same as above, except, that MOVE tries to calculate the dependants while reading the application list.

Edit>>Insert 2ndary file...

Adds a secondary files (99.cls- files), which will be converted by MOVE as an extension class. If you add several secondary files for the same class, those will be bundled in a single migration task, that will convert the methods defined in all secondary files into a single extension class.

Edit>>Remove from Project

Allows you to remove one entry from the migration task list.

Edit>>Change Package Name...

Allows you to change the destination package name. You may select multiple items for this purpose. The default package name is determined through the JavaClassEmitter, which works hand in hand with the `S2JEnvironment` method `defaultDestinationPackage`, which you may override for useful default values.

Edit>>Change State...

Allows to set a state of a migration task items (or many in one step). By doing so, you are able to track your things to do.

From the *Tools* menu you will actually start the generation process (*Tools>>Generate...*) or display statistics about the number of classes to convert (*Tools>>Statistics...*).

Additional hints for using the migration task list

When adding classes to the migration task list, MOVE will try to find existing instances and if none are found, tries to create them. This may result in lengthy operations. When running the generation, MOVE will try to open all ObjectStudio `Controller` objects before converting them. If you Controller to be converted is not prepared to be opened with the `openInDesignMode` method (cannot be edited in the ObjectStudio designer), you might run into debuggers. It is save to resume those debuggers to finish the generation.

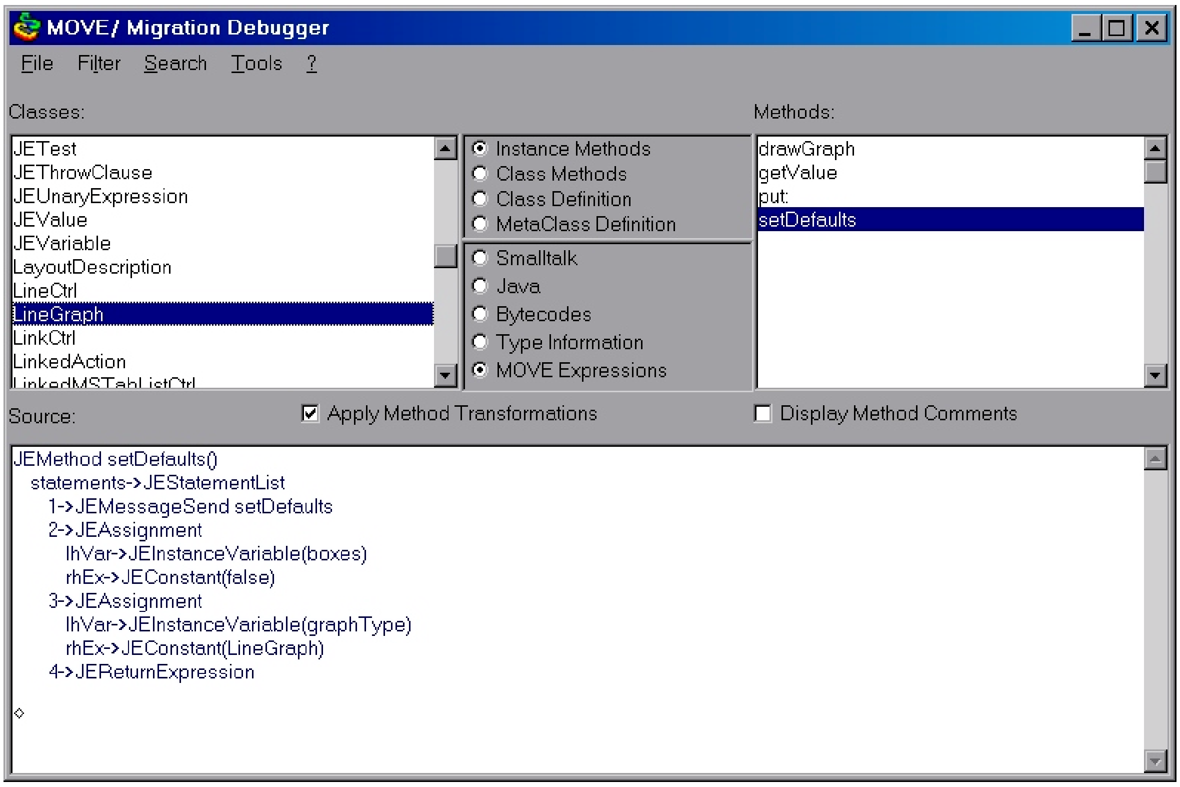
All converted Java sources converted are finally added to the current project of the MOVE Project Browser.

The MOVE/S2J Migration Debugger

Purpose

The Migration Debugger helps to maintain the MOVE/S2J conversion hint repository and to debug the syntax translator. It lets you browse Smalltalk classes and the corresponding converted Java source on the fly in a similar way as the ObjectStudio class browser. It can be used to compare the results produced by the MOVE/S2J source code translator with the Smalltalk source code and lets you debug and test your translation rules. Usually it is used in the process of customizing MOVE.

User Interface



[✓ Apply Method Transformations]

```
JEMethod setDefaults()
  statements->JEStatementList
    1->JEMessageSend setDefaults
    2->JEAssignment
      lhVar->JEInstanceVariable(boxes)
      rhEx->JEConstant(false)
    3->JEAssignment
      lhVar->JEInstanceVariable(graphType)
      rhEx->JEConstant(LineGraph)
    4->JEReturnExpression
```

It can display every method in 5 different ways:

- Smalltalk - Displays the method's original Source Code
- Java - Displays the translated source code. This allows you to directly test new rules you've defined for MOVE
- ByteCodes - This view is for advanced users.
- Type information - Displays the types as known to MOVE (in black color) or as being automatically derived applying type prediction (in blue color)
- MOVE Expression - This view helps you to understand the way MOVE structures the source.

The migration debugger can be used in the following ways.

- It can also be used to add additional conversion hints to the Smalltalk source to improve the cross compilation process. To do so, you switch the display option to *Type Information*. The conversion debugger will display the signature (type information) for methods it has analyzed. In Class Definition mode, the type information for class and instance variables of the class as known to MOVE/S2J is displayed. You can add additional type information by extending the Smalltalk Source code displayed here and select the menu point File>>Save Type Information. This will save the type information in the default *S2JTypePredictor* object. All known type infos (the result of a previous edit in the MOVE/S2JConversion debugger, or the result of using the MOVE/S2J Method Trace tool) are displayed using text color black. Type information as analyzed on the fly by automatic analysis (while you browse methods and classes in the repository browser) are displayed using the blue color.

Note: to make this information persistent for your next ObjectStudio session, you need to save the repository. This can be done by selecting the menu point File>>Save Repository.

- The repository browser lets you browse the known conversion hints of MOVE/S2J, that are stored n the MOVE/S2J repository.
- The classes and methods can be filtered to display only classes and methods, where type information is known to MOVE/Smalltalk to Java.
- This repository browser also helps you to find out which of the Conversion Rules apply to a particular method [UNDER CONSTRUCTION].
- For advanced users it contains a byte code debugging facility. ByteCodes are used by the *JavaCodeEmitter* to construct the syntax tree.

Using the Migration debugger

File>>Save Repository

Will save the MOVE/S2J Repository. Per default a filename of `gebit\s2j\s2j.repository` is assumed. You may backup this file from time to time to save your work with MOVE/S2J.

File>>Save Type Information

Will execute the source currently displayed in the editor source pane and evaluate the result as a Smalltalk object. The result will be registered in the *S2JTypePredictor* class under the reflector for the class selected in the class list and the method selected in the method list.

File>>Exit

Will close the browser

Filter>>Classes with Type Information

Will cause the class list to display only classes, for which conversion hints (*STClassReflectors*) for MOVE/S2J are known.

Filter>>Methods with Type Information

Will cause the method list to display only methods, for which conversion hints (*STMethodSignatures*) for MOVE/S2J are known.

Search>>Find class

Opens a dialog box, where you can type the name of a class to be found and selected in the class list. You may use wildcards (%) in search patterns.

Search>>Browse selected Method...

Opens the ObjectStudio class browser and selects the same class and the same method, that is selected currently in the repository browser.

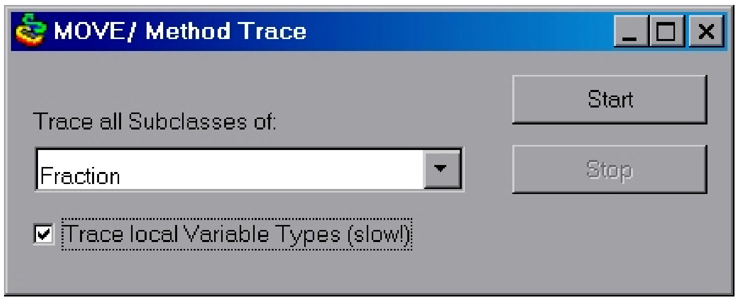
Tools>>Init Repository

Will reset all information in the MOVE/S2J Repository to an initial state. This option should be used only, if there are major problems with the conversion information known by MOVE/Smalltalk to Java.

Tools>>Analyze unique methods

This menu point should be used from time to time to make MOVE/S2J up to date with classes recently loaded into ObjectStudio.It will analyze all classes to find unique selector names. The information generated can be used for type prediction. Attention: selecting this facility may take a long time (about 1 minute) to complete, because it will cause an intensive analysis of the ObjectStudio class system.

The MOVE/S2J Method Trace Tool



Trace all Subclasses of:

Fraction

[✓] Trace local Variable Types (slow!)

Purpose

The Method Trace Tool records message sends in the system and analyzes type information based on the types of objects during run-time.

User Interface

This tool analyzes all messages in the system sent to objects being an instance of the class selected in the **Trace all subclasses of**: drop down list. It will analyze the types of concrete parameters and local variables, if the **Trace Local Variable Types** checkbox is selected. It will record the information found in the current *S2JTypePredictor* or will update the information about a particular method, if it finds out, type information must be relaxed. Running this tool on all subclasses of a hierarchy of classes migrated, can greatly improve the correctness of the type information generated by MOVE/S2J.

Tip: The Trace Local Variables option can be very slow and is recommended to be used not in combination with tracing all subclasses of Object.

Pressing the **Start** button will install a hook in the ObjectStudio system, that traces all messages. Pressing the **Stop** button will release that hook. After pressing the **Start** button, you may see significant performance degradation depending on the options selected.

To review the type information recorded, you can use the **MOVE/S2J Migration Debugger**, by using its Method Signatures option to review the known signatures in the classes of your interest. To store the information recorded for usage in later MOVE/S2J-sessions save the current MOVE/S2J Repository from the MOVE/S2J Repository browser.

The MOVE/S2J Repository

The MOVE/S2J repository contains information, that is useful to improve the Smalltalk to Java conversion process.

It contains type information for instance, class variables and for method arguments, return types and local variables.

It also contains other information for the MOVE/S2J type predictor, like all unique method selector names in the system and the name of the class implementing those methods.

It may contain additional custom information which can be attached by subclasses of the *S2JRepository* class, that are specific to your project.

The repository is usually placed in a file called `$OSTUDIO/gebit/s2j/s2j.repository`. The information stored here can be edited and improved, using the **MOVE/S2J Method Trace tool** and the **MOVE/S2J Repository Browser.**

Information stored in the repository can be extended or overridden by extending the method *binaryStorageReadComplete* in your custom subclass of *S2JTypePredictor*. This method will be called, when the default type predictory has been loaded from the repository. If you override it in a subclass, do not forget the corresponding super method. Instead of using the repository features, you might as well decide to extend the system programmatically by using this mechanism.

To store custom information in the repository, you should extend the *S2JRepository* class with the singleton pattern. In your subclass you should override the two methods *registerConversionHintsOn:* and *unpackConversionHintsFrom:* methods, to maintain a dictionary of additional information. In the *unpackConversionHintsFrom:* method you should not assume that the value of a particular key, that is defined by you exists in the dictionary, so your repository will be exchangeable with other vendors (or the original one). As key names we recommend symbols starting with your companies name like in the following example:

`#'gebit.primitiveTypeInfo'`.

III. MOVE/Smalltalk to Java - Code Translation

Overview

MOVE/S2Js source code translator is written entirely in Smalltalk. This chapter outlines how it works, because a good understanding of the algorithms used is mandatory to be able to customize the translation process. Please note, that the quality of the generated source code depends highly on the customization of the translator, so a good understanding of the following chapters will be the base for a successful migration effort. Customization helps you to:

- Define your project specific rules on how to convert your specific smalltalk constructs efficiently.
- Affect the source generation back-end to produce code compliant with your Java Style Guide.
- Affect the source generation to produce code to be used within your target Java Framework.

Smalltalk to Java Migration Challenges

To understand the process of source code cross compilation better, we will describe the most important differences between the Smalltalk and the Java languages first. For many of these issues MOVE/S2J provides a strategy, that solves that problem, some of those issues however cannot be solved without completely redesigning the whole architecture, so there is no automatic solution. Fortunately the features in the Smalltalk language causing those problems are rarely used by business applications directly.

Language differences

Types

The most often mentioned difference between Smalltalk and Java is, that Smalltalk does not require type declaration, while Java does. Declaring types helps the Java compiler to find errors in the source code, while it also restricts generity of the written code. MOVE/S2J uses different strategies for type prediction, that are documented in detail below, to add type declarations during the conversion process.

Primitive Values versus Objects

Java has a concept used frequently in Java applications called the primitive type. Primitive values are no real objects and can in particular not execute any methods. There are 7 primitive types defined in Java, which are frequently used in typical Java programs. For every of those primitive types, so called wrapper classes exist, that allow to treat primitive values as real objects. Unfortunately most of the expressions involving operators, like boolean or arithmetic expressions work on primitive values only. This means in particular, that the approach not to use primitive values at all in Java applications, but to use the corresponding wrapped objects instead, will create very clumsy unreadable Java code.

MOVE/S2J maps the Smalltalk Classes that correspond with those primitive types to the primitive version of the type. This causes arithmetic and boolean expressions to work naturally. In cases, where methods are sent to those objects, an expression is generated by MOVE/S2J, that will wrap the primitive value inside a real object and send the message to that object instead.

Example:

Smalltalk expression	*Generated Java expression*
`(i + 1) print`	`new Integer(i +  1).toString()`

Method Names and Operator overloading

To call methods, Smalltalk uses a notation where the message selector is divided into pieces for every argument of the method call. Java will have one identifier - the method name. In addition Smalltalk allows to use operator characters like + and ~ for method names, while Java has a defined set of operators, that can operate on primitive types only (see above) and cannot be changed. Also the set of allowed characters for identifiers and the list of reserved words differs between Smalltalk and Java.

MOVE/S2J will generate method names based on the Smalltalk name by concatenating the method selector fragments. Operators not being used in the context of an expression on primitive types are converted to appropriate messages. All illegal Java identifier characters are stripped from method and variable names and a check on a reserved word reserved in the Java language is finally performed to convert those to correct Java identifiers.

Examples:

Smalltalk names	Generated Java names
`extractStringsDelimiter:escape:`	`extractStringsDelimiterEscape`
`=`	`equals`
`==`	`equivalent`

Blocks

Smalltalk introduces the syntactic feature of blocks. Blocks are used in Smalltalk frequently to implement iteration statements or callback features for instance. A particular feature of the blocks as defined in ObjectStudio is, that they can reference the outer execution context (temporary variables and method arguments) of the method, where they are defined, even if that method is not actively executed any more. Particular blocks can be translated directly to standard Java control statements like if - then - else or to for- and while-loops.

MOVE/S2J converts blocks alternatively into iterator statements or using Inner Classes and the adapter pattern. The first approach works perfect also in combination with returns from the block (forced returns). As iterator statements in Java do not open a new execution context, variables of the outer context can be easily accessed. The second approach however might fail, if local variables of the surrounding method are accessed from a block or a forced return tries to escape the surrounding method context.

Example for Approach 1:

Smalltalk code	Generated Java code
`aColl do: [ :each |`	`Iterator tempI = aColl.iterator();`
`    each out.`	`while(tempI.hasMore()) {`
`].`	`    System.out.println(tempI.next());`
	`}`

Example for Approach 2:

Smalltalk code	Generated Java code
`aButton onEventDo: [`	`aButton.addEventListener(`
`    self performAction.`	`    new ActionEventListener() {`
`].`	`      public void actionPer-`
	`formed(ActionEvent e) {`
	`          performAction();`
	`      }`
	`    }`
	`);`

Example for Approach 3:

Smalltalk code	Generated Java code
`(aBoolean and:`	`if (aBoolean && anOtherBoolean) {`
`      [anOtherBoolean]) ifTrue: [`	`   yes();`
`   self yes`	`} else {`
`] ifFalse: [`	`   no();`
`   self no`	`}`
`].`	

Converting blocks accessing temporary variables or using forced returns can converted theoretically and in fact the following example on how they can be converted could be implemented using MOVE/S2J easily. However we feel, that the resulting source code, if you strictly follow that paradigm will be unmaintainable.

Smalltalk extended block usage

```
| a b c |
   a := 32.
   aCollection collect: [ :each |
      b := each + a.
      b == 88 ifTrue: [
         ^ self
      ].
      b
   ]
```

Java version with inner classes

```
int a;
int b;
InnerTempClass blockArg;
a = 32;
blockArg = new InnerTempClass();
blockArg.a = a;
try {
   aCollection collect(blockArg);
} catch(ForcedReturnException x) {
   return;
}
...
class InnerTempClass implements Value-
Functor {
   int a;
   public Object value(Object aValue) {
      int b;
      b = ((Inte-
ger)aValue).intValue()+a;
      if (b == 88) {
         throw new ForcedReturnExcep-
tion();
      }
      return new Integer(b);
   }
}
```

Class methods versus Static methods and Constructors

In Smalltalk every class is an object itself and has a corresponding meta class. Methods defined in classes (class methods) are therefor also dynamically dispatched depending on the receiver class object. In particular the use of super is allowed inside a class method. To construct a new object, usually some variation of the new method is used, which is finally delegated to Object>>new, the mother of all objects in Smalltalk. On the other hand, Java has no meta model. It has the concepts of static methods, that do not require an explicit instance of a class to be executed, but they are statically bound during compile time, there is no real receiver object, when they are executed and the usage of the super keyword inside a static method will not work. Object construction is performed through a special language concept - the constructors.

MOVE/S2J will convert class methods to static methods. As there are limitations in static methods as outlined above, this approach has some constraints, where it fails. Smalltalk Class Methods named *new...* are converted to constructors by MOVE/S2J.

Smalltalk code

```
tempObject := Object new.
tempFonts := Font availableFonts.
newName: anArg     "defined in Employee"
   ^ super new name: anArg.
```

Generated Java code

```
tempObject = new Object();
tempFonts = Font.availableFonts();
Employee(String anArg) {
    super();
    setName(anArg);
}
```

Global Variables and Pool Dictionaries

Java does not know neither of these concepts, they can be emulated however pretty simple. The interesting point is, how to emulate them in a way, that is a good Java design.

MOVE/S2J will create for every pool dictionary a class with the name of the pool dictionary and with static variables with name, value and type of the values in the pool dict. These classes will be created only on de mand however (see converting pool dictionaries) and not every time a class is converted, as pool dictionaries are shared by classes. References to pool values are treated like references to these static variables. Creating new pool variables on the fly is currently not supported, by using this concept. It could be implemented easily in the super class of all pool dict classes created. Global variables are treated as static variables in a class called *SmalltalkSystem* contained in the MOVE/Smalltalk to Java framework classes. This class defines some commonly used ENFIN global variables. To improve the overall design, this class does not contain all global variables, but only an important subset. Furthermore some variables were moved to different classes, like *Tab* and *CrLf*, which were moved to the *StringUtil* class. MOVE/S2J will create appropriately references in this case. For global variables not yet defined in any of the framework classes a *SmalltalkSystem.get* and a *SmalltalkSystem.atPut* method is generated.

Smalltalk	Generated Java code
`a := Space.`	`a = CharacterConstants.Space;`
`b := DirSeparator.`	`b = StringUtil.DirSeparator;`
`c := SomeGlobal.`	`c = SmalltalkSystem.get("SomeGlobal");`

nil

In Smalltalk *nil* is an object, which has in particular a class - UndefinedObject. Being an object it can receive messages like isNil and respond to those messages. Java has the concept of a *null* object reference. This concept allows to test, whether a variable references an object or not. In particular sending a message like toString() to a null reference will always cause an exception in the application.
MOVE/S2J converts the most important messages isNil, isNotNil... sent to a nil object to appropriate test expressions. References of nil directly are converted to references of null.

Example:

Smalltalk code	Generated Java code
`anObject isNil ifTrue: [...`	`if (anObject == null) { ...`

Unconvertable language features / *become:* and *doesNotUnderstand:*

Some of the features of the Smalltalk language cannot be implemented or even emulated with the Smalltalk language at all. The definition of a doesNotUnderstand: method in a particular class will have no effect in Java, because per definition all Java messages will be understood, because this is guaranteed based on type information by the Java compiler already. Also the transformation of object references with become: and becomeOneWith: will very likely never be implemented in the Java language due to its typed nature. As such features are completely missing, there is no good strategy to convert source code using them.
One of the typical cases, where those features are used in business applications, is the implementation of a *proxy* mechanism. To implement Proxies in Java, you would usually use one of the following strategies: proxy objects can be put into containers, that will always be evaluated by sending for instance a *getValue* method, before referencing the contents of that container, or you can put exit methods inside every getter methods.
Examples:

Smalltalk code using a proxy	Java version to emulate proxy nature
`" This method returns the color of a`	`// This method returns a color`
`..." color`	`public Color getColor() {`
`   ^ color. "color can be a proxy"`	`   resolve(); //This might load the`
	`proxy`
	`   return color;`
	`}`

Once you have decided on the pattern to use, you can define rules in MOVE/S2J that will generate the appropriate code to implement it.

Incorporation of new language features / Exceptions...

Java offers compared to Smalltalk new language features. A good Java program will use those features of course. One example for such a feature is, that Java provides in contrast to ObjectStudio Smalltalk exceptions. Other Smalltalk dialects will implement exceptions using messages, which is different to using a language feature, so the following strategy can be applied as well easily to convert this mismatch.

MOVE/S2J can incorporate exceptions into the generated source code, if the "pattern" on how exceptions are to be incorporated is well defined. This conversion pattern can be implemented, using a conversion rule, that analyses the statement tree.

Example: Assume, exceptions were emulated checking for error return objects in method calls explicitly. If an error is found, some optional error handling is done, than the error object is returned, otherwise processing continues. This pattern for error checking can be converted by MOVE/S2J to use exceptions instead as outlined in the examples below.

Smalltalk error handling	Java error handling with Exceptions

`myMethod`
`   | error |`
`   error := self callTransaction.`
`   error isTransactionError ifTrue: [`
`      ^ error`
`   ].`
`   ....`
	`void myMethod() throws TransactionErro-`
	`rException {`
	`   callTransaction();`
	`   ...`

`myMethod2`
`   | error |`
`   error := self callTransaction.`
`   error isTransactionError ifTrue: [`
`     self handleError.`
`     ^ error.`
`   ] ifFalse: [`
`     self continueProcessing.`
`   ].`
	`void myMethod2() throws TransactionErro-`
	`rException {`
	`   try {`
	`      callTransaction();`
	`      continueProcessing();`
	`   } catch(TransactionErrorException e)`
	`{`
	`      handleError();`
	`      throw e;`
	`   }`

Base Class Incompabilities

Different protocols

An easy incompatibility lies in the fact, that some (many) methods and classes in Smalltalk have a different name, than the corresponding method / class in Java. MOVE/S2J has a big repository of translation patterns, that will simply convert the Smalltalk version of a protocol to the corresponding Java construct. The set of translation patterns can be easily extended. This must not necessarily be a message send at all, as some of the examples below demonstrate.

Typical Smalltalk methods	Java pendants generated by MOVE/S2J
`anObject asString`	`anObject.toString()`
`aDict at: 12 put: 'Hello'`	`aDict.put(new Integer(12), 'Hello')`
`anObject isKindOf: Error`	`anObject instanceof ErrorException`
`anObject asciiValue`	`Character.getNumericValue(anObject)`

Incomplete protocols / not existing classes

Some features of Smalltalk classes are not implemented in Smalltalk. The ObjectStudio *Date* or *String* classes for instance defines a whole bunch of methods not found in the *java.util.Date* or *java.lang.String* classes. In particular the Date methods should be implemented in Java by going through the use of the Java Calendar class. There are two alternatives to workaround this issue:

- The missing functionality can be inlined by calling the appropriate methods in other classes respectively. This approach makes the resulting source code less readable but bases the conversion result on the default Java JDK library.
- An extension Java class can be provided, that implements the missing functionality instead. This is MOVE/S2J's approach, though it requires additional classes to be used on top of the JDK standard packages. MOVE/S2J provides an additional base framework, that is shipped with source code and can be extended. In the scenario, where an extension is needed, the extension class is used instead. The following pattern is applied, if the class cannot be extended (like *java.lang.String* or *java.lang.Object*). In those cases a class *Classname*Util is defined, that defines static methods implementing the functionality with the first argument being the receiver object.

Example:

Smalltalk base class usage	MOVE/S2J Java extension class usage
`aString like: '%hello'`	`StringUtil.like(aString, "%hello")`

`aDate monthsApart: 2`	`de.gebit.enfin.base.Date aDate;`
	`aDate.monthsApart(2);`

Semantic Differences / Strings, Collections

Some of the core classes differ in semantic between Smalltalk and Java. Two typical differences are, that String objects are read-only in Java and cannot be changed while they can be changed in Smalltalk and that the elements of collections in Smalltalk can accessed with a 1-based index, while the index is 0-based in Java.

Currently, there are no solutions to these issues in MOVE/S2J, we plan however to implement the following strategies:

- In contexts where collections are accessed, the parameters of accessing methods will be decremented in the conversion. When collection sizes are checked, the check is also changed appropriately.
- In contexts where Strings are changed in the Smalltalk code, they will be converted to Java StringBuffer objects or there type is completely changed to be a StringBuffer.

Algorithm of the Source Code Translator

The following algorithm describes the important steps in translating a Smalltalk class:

1. The class emitter will analyze all instance and class methods of the target class by inspecting the byte codes of those methods.
2. Based on the analysis it builds an expression tree for each method. The objects in that tree are instances of subclasses from the *JENode* class, the result of the method analysis itself is a *JEMethod* object. The class used to perform that analysis is the *JavaCodeEmitter* class, that can analyze Smalltalk *Block*s, *QuickMethod*s and *CompiledMethod*s. The tree generated consists of nodes, that can be directly stored as Java source code (Method *storeJavaOn:*).
   Note: Incompatible Smalltalk syntax nodes like blocks are currently translated to appropriate Java constructs directly. Ideally a Smalltalk syntax tree would be created first, that is transformed later into a corresponding Java tree.
3. At the end of the analysis, the code emitter tries to predict the types based on expression evaluation and the information gathered in the *S2JTypePredictor* class. This class implements a unique selector match strategy based on unique selector names found in the system. This strategy could be extended to collect more type information.
4. After type prediction is complete, the nodes of the tree are converted by executing the method *applyTransformationRule*. This will for some node types result in a call to the *convert:* method in a concrete subclass of *S2JNodeTransformationRule*. See below for a description on how to tune the conversion process.
5. Finally the source code is generated by calling the *storeJavaOn:* method in the *JavaClassEmitter* object.

MOVE/Smalltalk to Java Customization - an Overview

MOVE/S2J is shipped with Smalltalk source code, so it could be customized by simply changing that source code / writing secondary files etc.. The MOVE/S2J classes are designed however to allow for flexible customization using the *Singleton* design pattern. This pattern is described in the following section along with an example on how to use it to customize the migration process.

Many classes in MOVE/S2J are implemented as a singleton. This means a single instance of such classes exists and can be referred to by *ClassName default*. To return the current environment settings for instance, you can use the following statement: *S2JEnvironment default*. To customize the singleton objects, you can create your custom subclass of such an object and create one instance of that subclass to the default class variable of the super class. From that point on all MOVE/S2J classes will use your new default implementation of that singleton instead.

We recommend to put your custom classes in the directory: `gebit\s2j\custom\projectName`.

The following example will define a custom environment for MOVE/S2J, that defines a different class path to be used by MOVE/S2J's project browser.

```
S2JEnvironment subclass: #S2JCustomEnvironment
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: '' !

! S2JCustomEnvironment class methods!

initialize
    default := MyEnvironment new.
!
!

!S2JCustomEnvironment instance methods!

classPath
    ^ System environmentAt: #CLASSPATH
!
...
```

Why is it, we use this feature rather than a property file or something similar for the customization? The reason is: it is much more flexible. This approach is not limited to change attribute names, but can define additional behavior like a new rule to be applied (a method to be sent) to every *JEMethodNode* during the migration project to change that node on the fly.

Implementing a different translation strategy

MOVE/S2J converts Smalltalk classes through the usage of a *S2JConverter* class. The *S2JConverter* makes use of one or several *JavaClassEmitter* classes to generate one up to n Java classes for every Smalltalk class. The following example demonstrates how this feature can be used. Every class will be asked by the MOVE/S2J tools for the specific *S2JConverter* to do the conversion with the *getS2JConverter* method. Override this method along with the implementation of a specific Converter class allows to implement a specific strategy. S2JConverters may also be used to convert "other things" than classes. MOVE provides the following special S2JConverters (in addition to the standard class converter):

`S2JSecondaryFilesConverter`

This converter will create an *extension class* for every method defined in a list of secondary files for a particular ObjectStudio class. The *extension class* will be named `ClassName`*`Util`*, an extension for additional functionalities to the core class `Date` would be for instance called `DateUtil`. Every method created in the extension class will be a `static` method and will be passed the receiver as a first argument.

`S2JPoolDictionaryConverter`

Will create a Java interface for a pool dictionary, defining all the constants of the pool dictionary as static constants. This interface can be implemented by classes originally using the pool dictionary. *Note*: Although pool dictionaries are in theory writeable (which is not supported by this conversion mechanism of MOVE), they are hardly modified in typical Smalltalk applications.

`S2JInterfacPartConverter`

Will create several classes for each ObjectStudio `Controller` class: one mirroring the actual source code of the original controller and one for each form defined in the controller, which only has the source code to create the appropriate form items. The later classes can usually be edited by a Java GUI builder. The way they are created is determined by the currently selected Form code generator.

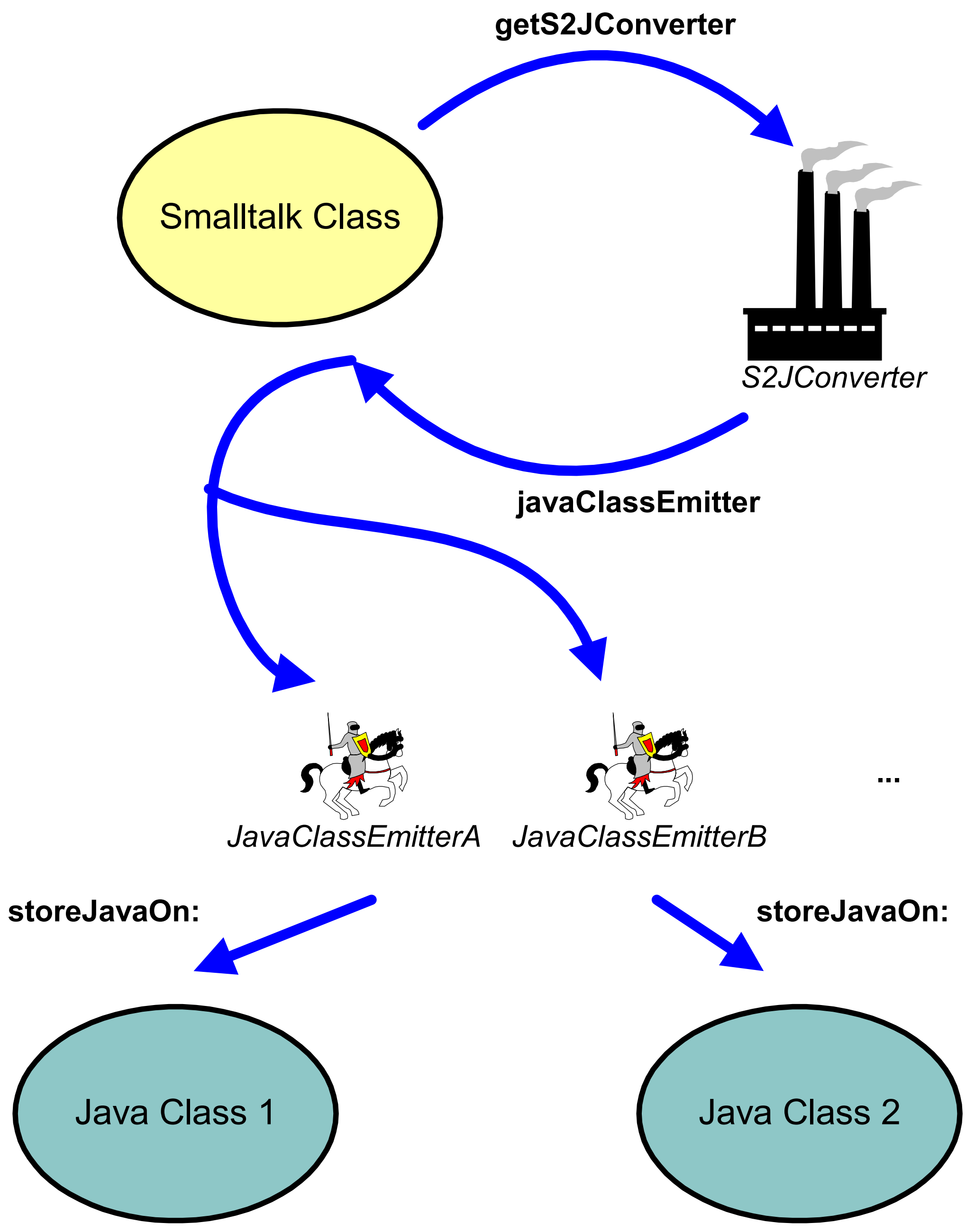
Assume somebody tries to implement the following style guide pattern. Good practice in Java is to use *interfaces* instead of *classes* in type declarations. To implement this style guide, one could generate an interface for every Smalltalk class with the name of the Smalltalk class and one Implementation class, called i.e. *BasicClassName* by writing a particular *S2JConverter*.

The *JavaClassEmitter* class is then responsible to implement one specific cross compilation strategy for a class. It is usually instantiated based on a source class to be translated or based on a prototype object of that class. The 2nd approach has the advantage, that MOVE/S2J's type analyzer has more additional hints on the type of instance variables by analyzing an existing object. You never instantiate the *JavaClassEmitter* directly, but use the method *javaClassEmitter* defined in Behavior, that (or *javaEmitterClass*) can be overridden to return a specific *JavaClassEmitter* for a particular set of classes.

As documented above, every class can define specialized *JavaClassEmitter* classes responsible to convert them. Currently there are the following *JavaClassEmitter* classes implemented and shipped with MOVE/S2J:

- JavaClassEmitter - default emitter for normal classes
- JavaInterfacePartClassEmitter - emitter for GUI classes
- JavaFormClassEmitter - emitter for ENFIN forms to corresponding Java panel subclasses
- JavaInterfaceForClassEmitter - emitter to emit the corresponding Java interface for a class
- JavaInterfaceForPoolDictionaryEmitter - emitter to create a Java interface for a pool dictionary.

The following picture demonstrates the message flow during the conversion process, that will finally generate the set of destination classes.



Tuning the translation process

The translation process can be configured by defining translation rules. A conversion rule basically takes a node from the expression tree (see above) and creates a new node. This will replace the other node in the expression tree and will be used finally when generating the Java source code.

- To define additional rules for message conversions, create a secondary file for the *S2JMessageNodeTransformationRule* class and create additional message in an *initialize* method conversion rules like the ones given in the following examples.
- Currently two rules are defined to transform the names of methods defined as being so called setter or getter methods according to the Java style guide, to name those methods *get(set)AttributeName*. To define additional rules to applied on the method object, create a secondary file for the *S2JMethodNodeTransformationRule* class and add additional rules similar to the ones added in the initialize method of the primary class file for this class.
- Currently rule classes are defined to transform message send nodes (*JEMessageSend*), method nodes (*JEMethod*) and complete statement lists (*JEStatementList*) and some other node types. To introduce new conversion types, override the appropriate *applyTransformationRule* method and introduce a new class, that does the conversion.
  Alternatively you can perform transformations in the *applyTransformationRule* method of the appropriate node directly (the *JEBinExpression* class does this currently). However we recommend to follow the 1st alternative to do the implementation as it provides more flexibility.
- The *S2JClassTransformationRule* singleton can be extended (see above), to define how Smalltalk classes are mapped to corresponding Java classes. By default mappings are defined for mapping between numeric types (Integer, Decimal...) to the corresponding Java (if available primitive) types like int, java.util.Decimal.

Examples for Node TransformationRules

The following message conversion rule converts all occurances of the method *class* sent to an object to the method name getClass.

```
S2JMessageNodeTransformationRule new: #class do: [ :msg |
   msg selector: #getClass
].
```

This will result in the following translation:

```
anObject class -> anObject.getClass()
```

The following more complex rule will convert all occurances of *isNil* to a comparison with null.

```
S2JMessageNodeTransformationRule new: #isNil do: [ :msg |
    JEBinExpr left: msg receiver op: #== right: nil.
].
```

This will result in the following translation:

```
anObject isNil -> (anObject == null)
```

The following rule will convert all occurances of *the* message *asciiValue* to be converted, but only in the case, where MOVE/S2J has recognized that the message receivers type is Character.

```
S2JmessageNodeTransformationRule new: #asciiValue
        condition: [ :msg |
            msg receiver typeName = #Character
        ] do: [ : msg |
            JEMessageSend
                receiver: (JEVariable new: 'Character')
                selector: #getNumericValue
                args: (Array with: msg receiver)
        ].
```

This will result in the following translation:

```
anObject asciiValue -> Character.getNumericValue(anObject)
```

Custom JavaClassEmitters

Custom JavaClassEmitters are usually useful for the following reasons:

- you may define your custom source code optimization depending on a target class. An example would be the replacement of all getter and setter methods with a special Java implementation version. Override the method `processJEMethod:withContext:` to add custom processing to the individual methods generated. A powerful way to process methods is provided through S2JTreeTransformer objects.
- you may add filters (override `acceptsMethod:` and `acceptsClassMethod:`) or generate additional methods on the Java side for a particular class (`emitInstanceMethodsOn:`)
- another useful feature is to define the standard import lists for the classes generated for a particular subclass (MOVE also supports changing import lists for a class dynamically). To do so, override the `initializeInstanceVariables` method and change the `importList` appropriately

S2JTreeTransformers

S2JTreeTransformers can be applied to every MOVE expression tree (usually to objects of the type JEMethod) to transform an expression tree. The following sample demonstrates, how to create a tree transformer, which will replace all occurrences of access to the particular instance variables (`controller` and `subject`) with an appropriate method call:

```
tempTransformer := S2JTreeTransformer new.
tempTransformer context: aContext.
tempTransformer on: JEInstanceVariable do: [ :each |
         each name = 'controller' ifTrue: [
                  self viewGetterMessage.
         ] ifFalse: [
                  each name = 'subject' ifTrue: [
                  self subjectGetterMessage.
                  ] ifFalse: [
                           each
                  ].
         ].
].
^ aJEMethod transformUsing: tempTransformer.
```

The transformer object has always an appropriate `JavaCodeEmitterContext` object assigned. This object is i.e. passed to the `processJEMethod:withContext:` method mentioned before. It has a couple of transformation rules, usually defined as a block to be executed for a particular node type and produce either

- the unchanged JENode object passed as a value
- a new transformed JENode object
- nil in which case, the node is eliminated in the result tree

Tuning Type Prediction

Currently MOVE/S2J performs type prediction based on the following different strategies.

- It tries to identify an *existing instance* of an object. If such exists, it inspects the types of the objects currently assigned to instance variables in that instance. Currently it always assumes those types are correct, though this approach fails of course in cases, where the actual type is not specific (Object).
- Based on *constants* and newly created objects (constructor expressions found in the source code) and their assignment to variables MOVE/S2J predicts the type of a variable.
- The *S2JTypePredictor* collects the information about method names used in single classes at startup time. When such a method is found in a message send in the source code, the receivers type is assigned to the class defining that type.
- The MOVE/S2J Event Trace tool can be used to analyze types during runtime.

- Finally every class can implement the *getSTSignatureAt:* method to return a *STMethodSignature* object for a particular method name. The default implementation in Behavior asks the *S2JTypePredictor* default instance, whether such a signature is defined. STMethodSignatures contain information about return type, type of arguments and types of local variables of a Smalltalk method and such extend Smalltalks reflection framework. The returned information must not be complete, i.e. may specify only the return type of a method. The type predictor stores a dictionary of STClassReflector objects, that could be extended with additional information to improve type prediction. This mechanism is the most reliable type prediction mechanism and helps to achieve always correct type prediction.

Custom Form Generators

Java has the promise "write once run anywhere". Surprise: there are still platform dependencies. One of these dependencies is the dependence on a GUI builder when creating forms visually. MOVE supports the generation of Java form classes from ENFIN controllers in a way, they can be edited visually in a GUI builder afterwards. The code generated for this purpose depends however on the specific tool used. By default MOVE supports the following Java tools by default:

- BEA's Java IDE VisualCafé (`VisualCafeCodeGenerator`)
- IBM's VisualAge for Java (`VAJavaCodeGenerator`)

You may however adapt MOVE to your tool of choice by writing a custom form generator. To do so, you will

- subclass the class `FormCodeGenerator`
- write you own implementation of the method `storeGeneratedCodeFor:`
- optionally redefine additional methods defined by the abstract base class implementation

To name your subclass, use the target tools name and append 'CodeGenerator'. A custom form generator for Inprises's Java IDE JBuilder for example, would be named *`JBuilder`*`CodeGenerator`. Following this naming convention is mandatory.
To make your subclass the default handler to generate the code for GUI forms, create a class initialize method in your custom subclass and set the class variable `defaultGeneratorClass` to be your class.

Customizing the formatting style and copyrights

To control certain aspects on how the produced source code MOVE/S2J makes use of the class *S2JFormattingStyle*. It uses the singleton object *S2JFormattingStyle default*, which can be overridden for customization reasons with the customization pattern described in this document.
The formatting properties controlled here are:

- expand tabs - whether tab characters should be expanded to white space
- tab size - the number of space characters used instead of one tab character
- copyright header - the copyright header to be created for every
- Several methods, that control the way in which generated Java methods are stored on the output stream. (To be completed).
- The order in which access modifiers (protected, final...) are stored.
- The order in which the Java class sections are stored and section headers generated.

IV. MOVE/Smalltalk to Java - Framework classes

The MOVE/Smalltalk to Java Framework classes are based on Swing and GEBIT's framework TREND/Framework for Java. They help MOVE/S2J to simplify the process of plumbing the generated code into the standard Java API. Ideally the extension classes are not necessary, but the generated source code relies completely on the standard Java API. You might want to tune the translation process to really do so, but in this version of MOVE/S2J some small extensions are needed. The following packages in the MOVE/S2J framework are of particular interest:

`de.gebit.s2j.smalltalk.base`

contains some extensions to classes also available in Java like Point (java.awt.Point), but with a limited protocol set supported. This package also includes some classes not available in Java at all, like Message or some utilities for handling often used ObjectStudio paradigms (like Smalltalk reflection) not convertable in a useful way.

`java12.util, java12.lang`

preview packages containing collection classes available starting from JDK 1.1. As soon, as you move your project from using JDK 1.1 to using JDK 1.2, you may remove these classes and the corresponding references. Currently most ObjectStudio collections and the corresponding protocol is mapped directly to the Java 2 collection library.

`de.gebit.smalltalk.s2j.gui`

contains wrappers for the ENFIN GUI classes. These classes are required only if you are migrating user interfaces.

`de.gebit.smalltalk.s2j.gui.fitem`

contains some concrete implementations for ENFIN FormItems.

`de.gebit.trend.collection`

These extensions of the standard Java Collection framework are also part of TREND and implement functionality of Smalltalk collections not available in the core Java package.

`de.gebit.trend.*`

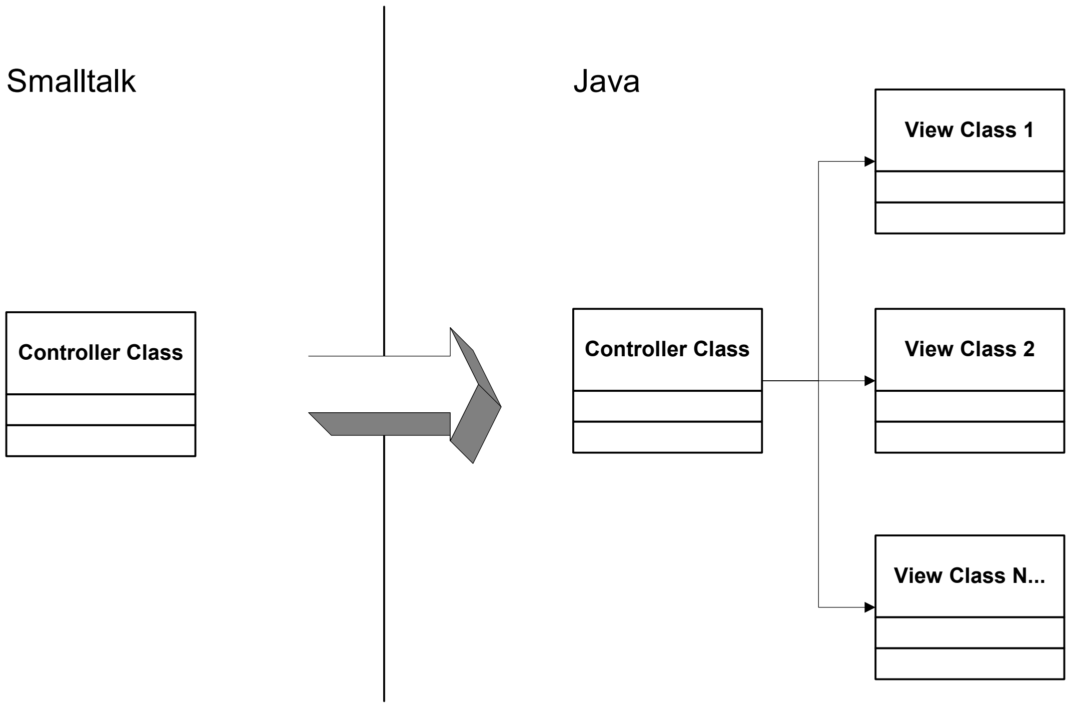
other packages from the TREND framework, which are used for dealing with

- special form items
- reflection (invocation of methods, ...)
- exception handling
- multi language support for user interfaces

Converting User Interfaces in MOVE/S2J

MOVE/S2J converts each ObjectStudio Controller and InterfaceComponent (currently not fully implemented) to several Java classes:

- For every top level form contained in the Controller it will create a separate View class, that contains the definitions of the FormItems and Forms and can (eventually) being edited by another tool such as Symantec Cafe. Every View class is named with a String, that is the concatenation of the Controller and the Forms name.
- All ControllerItem related information and methods implemented in a Controller are saved to a Controller class. The class will get the same name, the controller has.



This is the default behavior of MOVE/S2J, but it can be customized by subclassing the *S2JInterfacePart* converter class.

For all ObjectStudio Form and FormItem objects, MOVE/S2J will lookup the name of a concrete Java implementation class in the *S2JClassTranformationRule* singleton object. Default implementations for some Forms and FormItems are provided. Most of the implementations are currently based on Swing controls. In cases, where no corresponding Swing control is available (i.e. FormMaskField, FormDate, chart objects, Notebook,...) a part of GEBIT's framework TREND is shipped with MOVE/S2J and a mapping to the components implemented herein are used. You are however free to provide another wrapping layer to i.e. map to the KL Group beans or another vendors beans.

The view related properties of the FormItems such as formatting information (Color, fonts, etc.) are stored in a Java Beans compliant way with the Form class. All event listeners are currently registered in the Controller class but this registration may eventually be moved to the View classes as well.

Customizing the used GUI framework

MOVE/S2J does not require you to use a specific set of GUI controls such as the JFC (Swing)-Classes. It provides an abstraction layer to make it possible to plug the GUI class library of your choice. The default library ships based on AWT 1.1 and the GEBIT business framework TREND. The abstraction of the GUI class library works through the applicance of two concepts:

The protocol, that is required by MOVE/S2J to implement the GUI behavior is defined through Java interfaces. Those interfaces are defined in classes with a name, matching the Smalltalk class name in the package *de.gebit.s2j.smalltalk.gui*. By writing a subclass of a GUI component from a library of your choice, that implements the corresponding interface, you can use this class instead of the default class provided by MOVE/S2J.

The corresponding implementation classes are placed in the package
`de.gebit.s2j.smalltalk.gui.fitem`.

The actual mapping between the ENFIN form items and the concrete implementation of a corresponding Java component is defined in the *S2JClassTransformationRule* class. You can change all the mappings defined herein.

Currently the following form items are supported:

- `FormToolbar`
- `FormNoteBook`
- `FormBarChart`
- `FormLineGraph`
- `FormList`
- `FormRadioButton`
- `FormMLString`
- `FormDropList`
- `FormNumericSpinButton`
- `FormMaskField`
- `FormNumber`
- `FormCheckBox`
- `FormMSTabList`
- `FormPassword`
- `FormTabList`
- `FormTopicBox`
- `FormPieChart`
- `FormCheckList`
- `FormMSList`
- `FormProgressBar`
- `FormPopupMenu`
- `FormDropCombo`
- `FormSpinButton`
- `FormDate`
- `FormStatic`
- `FormStatusLine`
- `FormSlider`
- `FormMSTabList`
- `FormButtonList`
- `FormScatterGraph`
- `FormLine`
- `FormValueSet`
- `FormCombo`
- `FormTreeView`
- `FormSubMenu`
- `FormRect`
- `FormButton`
- `FormString`
- `FormBitmap`
- `FormBitmap`

Anforderungsanalyse • Architekturberatung • Implementierung • Projektrealisierung

XML J2EE OOAD OS/390 Java SAP EAI B2B EJB .NET

Individuelle Lösungen • Projektleitung • Produktentwicklung • Coaching und Training

www.gebit.de

GEBIT Solutions GmbH		Geschäftsstelle Düsseldorf		Geschäftsstelle Stuttgart	
Koenigsallee 75 b		Hammer Straße 19		Liebknechtstraße 33	
14193 Berlin		40219 Düsseldorf		70565 Stuttgart	
Telefon:	(030) 896 663 - 00	Telefon:	(0211) 513 414 - 0	Telefon:	(0711) 781 165 - 0
Fax:	(030) 896 663 - 36	Fax:	(0211) 513 414 - 444	Fax:	(0711) 781 165 - 5
E-Mail:	berlin@gebit.de	E-Mail:	duesseldorf@gebit.de	E-Mail:	stuttgart@gebit.de