PROJECTWORK 2

Accapto

DOCUMENTATION

Anja Bergmann
Johanna Kirchmaier

ITM14, WS16

Tutor: Elmar Krainz

Februar 2017

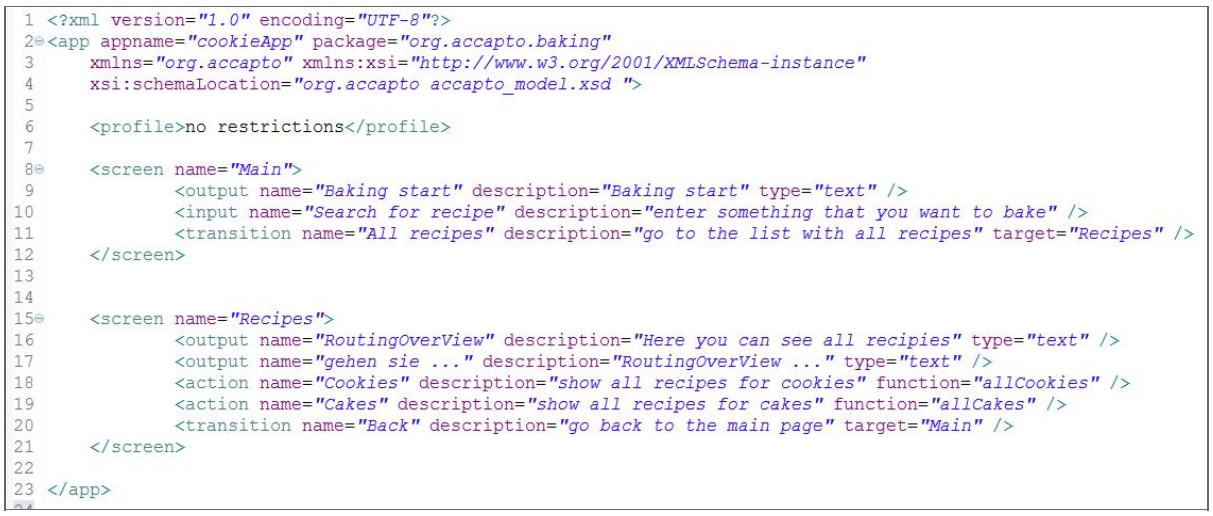
Index

How to get the project into Eclipse

- Download from https://github.com/gitTARDIS/Accapto
- Import into eclipse
- Add Libraries to build path
    - Right click on project
    - Click on _Properties_
    - Click on _Java Build Path_
    - Go to tab _Libraries_
    - Click on _Add JARs_
    - Add following files:
        - lib/apache-freemarker-2.3.25-incubating-bin/freemarker.jar
        - lib/commons-cli-1.3.1/commons-cli-1.3.1.jar
        - lib/commons-io-2.5/commons-io-2.5.jar
- Add javadoc for external libs:
    - Right click on project
    - Click on _Properties_
    - Click on _Java Build Path_
    - Go to tab _Libraries_
    - Expand the particular library and double click on _Javadoc location_
    - The locations for the particular javadoc files are:
        - lib/commons-cli-1.3.1/commons-cli-1.3.1-javadoc.jar
        - lib/commons-io-2.5/commons-io-2.5-javadoc.jar
    - For freemarker there is no javadoc file available; therefore add lib/apache-freemarker-2.3.25-incubating-src as Source attachment
- Run Accapto
    - Right click on project
    - Click on _Properties_
    - Click on _Run/Debug Settings_
    - Select _New..._
    - Select _Java Application_
    - Add org.accapto.tool.Accapto as Main class
    - Go to tab _Arguments_
    - Add desired arguments. To make Accapto print a list of all possible arguments, run it with -h or --help

How to write an input file

To use Accapto, an XML input file is needed. The xsd can be found at modelxml/accapto_model.xsd



```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <app appname="cookieApp" package="org.accapto.baking"
3      xmlns="org.accapto" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="org.accapto accapto_model.xsd ">
5
6      <profile>no restrictions</profile>
7
8      <screen name="Main">
9              <output name="Baking start" description="Baking start" type="text" />
10             <input name="Search for recipe" description="enter something that you want to bake" />
11             <transition name="All recipes" description="go to the list with all recipes" target="Recipes" />
12     </screen>
13
14
15     <screen name="Recipes">
16             <output name="RoutingOverView" description="Here you can see all recipies" type="text" />
17             <output name="gehen sie ..." description="RoutingOverView ..." type="text" />
18             <action name="Cookies" description="show all recipes for cookies" function="allCookies" />
19             <action name="Cakes" description="show all recipes for cakes" function="allCakes" />
20             <transition name="Back" description="go back to the main page" target="Main" />
21     </screen>
22
23 </app>
```

Even without reading the xsd, the file structure is easy to understand. Line 2 contains the **name** and the **package** of the app. Line 6 specifies the **profile**. The possible values are “no restrictions”, “blind”, “easy read”, and “easy touch”. The appearance of the app depends on this setting (not yet in this version, but as soon as accessibility features are included). The **screen** elements define the screens that the app will have. For every screen, an Activity.java file and a layout.xml file is created. Screens can contain four types of elements: **output**, **input**, **action** and **transition**. Output elements are simple text views, input elements consist of a text view and an input field. For every action element, a button will be created in the layout and an onClick function will be added to it. The activity file for the appropriate screen will contain a stub method with the appropriate name so that the code for the business logic can be added easily at the right place by the developer. The same applies to the transition elements. However, the code for an intent is already added to the function.

How to use Accapto (arguments)

Accapto can be called using one or more of the following options:

```
usage: accapto
 -?,--help           shows usage of accapto and possible options
 -h,--help           shows usage of accapto and possible options
 -i,--input <arg>    input file path (must be of type .xml)
 -o,--output <arg>   path where the app will be created (default: accapto
                     directory)
 -v,--verbose        print additional status information
```

This dialog can be displayed with the aid of the options **-?**, **-h** or **--help**, but would also be shown if only Accapto was started. The option **-i** or **--input <arg>** requires the path to an existing XML file, in which the presettings for the new app project are defined. Furthermore, the option **-o** or **--output <arg>** makes it possible to insert the directory path where the new app project should get created. The default location for that is the same directory in which the Accapto programme already is located. To get more detailed information about the processes of the tool when creating a new app, the option **-v** or **--verbose** can be included. This returns additional status information about the programme flow. For more logging information, just check the created Logger text file in the app project called accapto.log.

Documentation

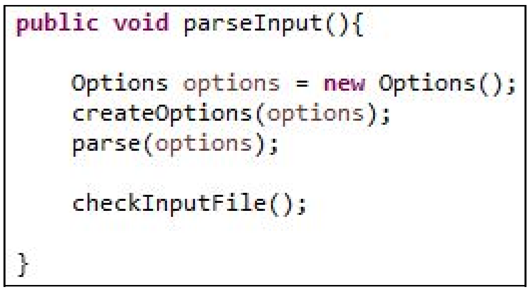
org.accapto.AccaptoMain

org.accapto.AccaptoMain contains the main class of the project. There are three other classes that need to be imported to start the programme: **java.io.File**, **org.accapto.helper.InputParser** and **org.accapto.helper.Logger**. First and foremost, an instance of the class InputParser is created and the arguments that are passed through the command line are handed over to the InputParser.
The main class now fetches the **File** ***inputFile*** and the **Logger** ***logger*** object that the inputParser created, because these objects will be needed in other classes as well.
Now a **ModelParser** object ***parser*** is created with the ***inputFile*** and the ***logger*** variables in its argument list. Then, the method ***parser.parseDSL()*** is called to parse the XML file and create a model of the app.
Afterwards an **AppScaffolder** is created. This object needs the app model that the ModelParser just created, the ***logger***, and the path where the app should be created. The method **Appscaffolder*****.generate()*** now generates the app scaffold.

Each of the above mentioned classes are described more in detail in the following.

org.accapto.helper.InputParser

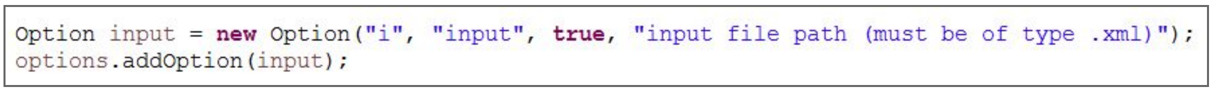
org.accapto.helper.InputParser parses passed arguments. This happens via the method ***parseInput()***.



```
public void parseInput(){

    Options options = new Options();
    createOptions(options);
    parse(options);

    checkInputFile();

}
```

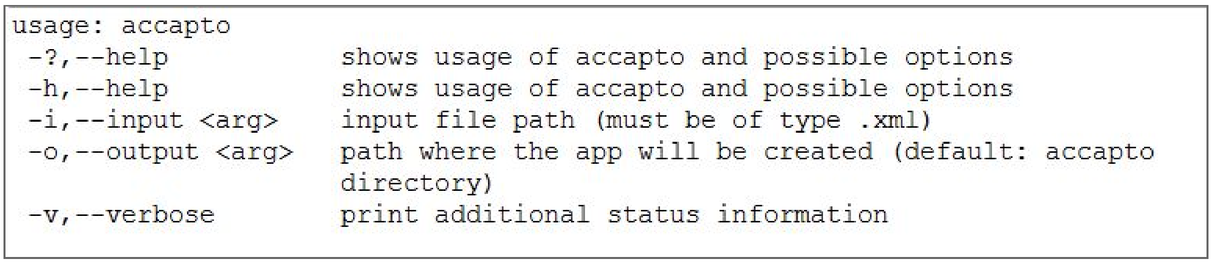
This method specifies which arguments can be used at the Accapto call and parses them. This is done by using the command line interface by Apache, **org.apache.commons.cli**.

Creating a new option requires the following lines …



```
Option input = new Option("i", "input", true, "input file path (must be of type .xml)");
options.addOption(input);
```

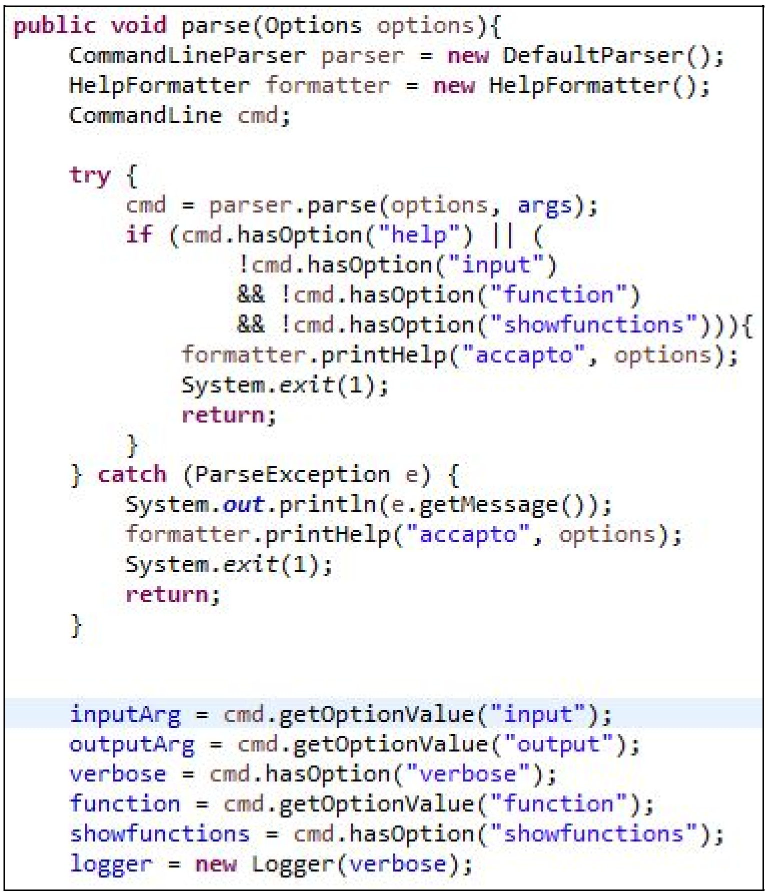
... where the first and second parameters are a character and a string that represent the option on the command line. Either the character with a minus in front of it or the string with two minuses can be used on the command line. The third parameter is of type boolean and defines whether the argument has an additional arg (e.g. for --input some/input/file.xml the param would be true, for --verbose it would be false). The fourth parameter is a string that describes the option and will be shown to the user if the help is printed. This can be done via the method ***formatter.printHelp("accapto", options)*** and will produce a console output like the following:



```
usage: accapto
 -?,--help            shows usage of accapto and possible options
 -h,--help            shows usage of accapto and possible options
 -i,--input <arg>     input file path (must be of type .xml)
 -o,--output <arg>    path where the app will be created (default: accapto
                      directory)
 -v,--verbose         print additional status information
```

Afterwards it is set whether the option input is absolutely necessary and eventually, this new created option input is added via ***addOption()*** to the **Options** object.

The method *parse()* distributes the taken arguments to the foreseen class variables.



```
public void parse(Options options){
    CommandLineParser parser = new DefaultParser();
    HelpFormatter formatter = new HelpFormatter();
    CommandLine cmd;

    try {
        cmd = parser.parse(options, args);
        if (cmd.hasOption("help") || (
                !cmd.hasOption("input")
                && !cmd.hasOption("function")
                && !cmd.hasOption("showfunctions"))){
            formatter.printHelp("accapto", options);
            System.exit(1);
            return;
        }
    } catch (ParseException e) {
        System.out.println(e.getMessage());
        formatter.printHelp("accapto", options);
        System.exit(1);
        return;
    }


    inputArg = cmd.getOptionValue("input");
    outputArg = cmd.getOptionValue("output");
    verbose = cmd.hasOption("verbose");
    function = cmd.getOptionValue("function");
    showfunctions = cmd.hasOption("showfunctions");
    logger = new Logger(verbose);
```

If the parsing of the input went well (i. e. no invalid arguments, all required arguments present, etc.), ***checkInputFile()*** checks if the input file is of type xml and can be found in the file system. If this is not the case, the programme stops with an error message that tells the user to check if the input file really exists at the given path.

org.accapto.helper.Logger

org.accapto.helper.Logger is able to write messages of the current status information into a log file and to the terminal. Since the log file **accapto.log** is not present until the app folder is created, the log messages created before that instant are stored in an ArrayList and added to the log file as soon as it is created.
If Accapto is called with the option --verbose, the method ***log(String message)*** will write messages to the log file as well as the standard.out, while ***onlyFile(String message)*** only writes to the log file in any case. ***logErr(String message)*** will write the message to the log file and to standard.err, independently of the --verbose setting.
For every log file entry, a time stamp is added to the message.

org.accapto.tool.ModelParser

The main purpose of this class is to create a Java Model from the given XML input file. First, the xml file is validated against the xsd file (located at modelxml/accapto_model.xsd) via ***validateAgainstXSD()***. If the file is invalid, the programme stops and prints an error message that tells the user to check the input file. If the file is valid, ***parseDSL()*** creates a model of the app, using the classes located in the package **org.accapto.model**. These classes represent elementes of the input xml file.

org.accapto.tool.AppScaffolder

If the parsing went well, **AccaptoMain** creates an instance of the class **org.accapto.tool.AppScaffolder** and calls the Method **AppScaffolder.*generate()*** which creates the basic structure for the Android Gradle Project. The method ***copyAppFolder()*** copies a basic app folder (located at Templates/DefaultApp) to the output location using the **org.apache.commons.io** library. ***createPackageFolders()*** then adds the folders for the packages that are needed.

public void copyAppFolder()

The main purpose of this method is to copy the template application folder to the given app directory, where ***srcDir*** defines the source directory, namely the template folder, and ***destDir*** defines the destination directory of the new app project. The method ***copyDirectory(srcDir, destDir)*** from **org.apache.commons.io.FileUtils** copies the specified directory and all its child directories and files to the specified destination. This is done within a try/catch-block, since IOExceptions might occur. In that case, the error is printed and the programme gets terminated with *System.exit(1)*.

private void createPackageFolders()

Like the name already suggests, this method creates the package structures for the new app project, to be exact the path structures ‘app/src/main/java’ and ‘app/src/test/java’. This is done with the variables ***activityPath*** and ***testPath*** and by the **java.io.File** method ***mkdirs()***. Since the packages are specified with dots between them (like this.is.a.package), the dots have to be replaced by file separators with the method ***replace()***.

public void generate()

This is the main method of the AppScaffolder class and generates a project that contains all essential files and directories for a basic Android application without the business logic. When this method gets called, firstly it is checked with an if-statement whether the names of the app and the package are known and not null. If this is the case, the two above mentioned methods ***copyAppFolder()*** and ***createPackageFolders()*** are executed. Moreover, the file accapto.log is created via ***logger.initLog(appPath)*** in the directory of the new app project. To add the necessary files files to the project structure (for instance, the manifest, the activities and layouts), a new **org.accapto.templating.Templating** object called ***templating*** is created and the variables ***app, logger, appPath*** and ***activityPath*** are passed as arguments. Eventually, the method *startTemplating()* is called (which is described in more detail at **Templating**).

org.accapto.tool.templating.Templating

The class Templating creates the domain specific scaffold by a Java model and eventually the files for the new app project. For this purpose, various variables are necessary, which are described in the following:

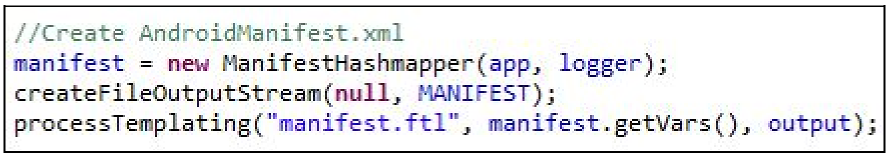
- the Configuration ***cfg*** for the FreeMarker template engine configuration,
- the AppType ***app*** for the app model,
- the PrintWriter ***output*** which acts as a Writer to write strings to files,
- the Logger object ***logger*** to write log messages to sys.out and to the log file,
- the Path ***appdir*** which is the directory where the app scaffold is located,
- the Path ***activitydir*** for the directory inside the app scaffold folder where the activity files will be located,
- the Path ***manifestdir*** which stands for the directory inside the app scaffold where the manifest file will be located,
- the ActivityHashmapper ***activity***, which is a Hashmapper object for creating activity files (for more information, take a look at the class **ActivityHashmapper**),
- the LayoutHashmapper ***layout***, which is another Hashmapper object for creating layout files (for more information, take a look at the class **LayoutHashmapper**) and last but not least,
- the ManifestHashmapper ***manifest*** for another Hashmapper object for creating manifest files (for more information, take a look at the class **ManifestHashmapper**).

In the constructor of Templating the configuration of the FreeMarker template engine is created and the passed parameters are stored in the appropriate class variables.

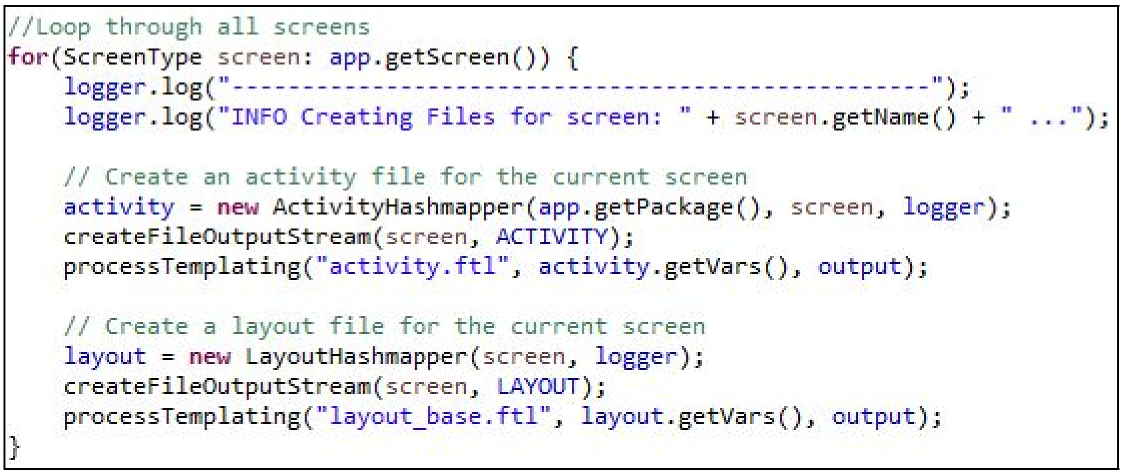
Methods

Before the method ***startTemplating()*** in this class can be fully explained, it is important to mention that this method calls two other methods that are implemented in this class. On the one hand ***processTemplating()*** and on the other hand ***createFileOutputStream()***. The first one starts the FreeMarker template engine for the specified template and a templating model. The latter one creates an OutputStreamWriter for the different files which are specified by the type. The **ScreenType** ***screen*** from the Java model and the String ***type*** for the type of the file are the arguments of the method that get passed. At first, a new File ***file*** is initialised with the value *null* and then - depending on the type which can be either an activity, a layout or a manifest - the ***file*** gets the correct path name assigned via a switch statement. At the end, a try-catch block creates the OutputStreamWriter for the ***file*** and catches a FileNotFoundException.

The ***startTemplating()*** method generates the AndroidManifest.xml file, the activity files and the layout files. As mentioned before, the above explained methods are called in this method multiple times. Firstly, the AndroidManifest.xml is created with an instance of the **ManifestHashmapper** called ***manifest***.


```
//Create AndroidManifest.xml
manifest = new ManifestHashmapper(app, logger);
createFileOutputStream(null, MANIFEST);
processTemplating("manifest.ftl", manifest.getVars(), output);
```

For the activity and layout files a loop through all screens is necessary, since every screen requires an activity and a layout. The activity file as well as the layout file for the current screen are generated in this loop, so that in the end all files are generated.


```
//Loop through all screens
for(ScreenType screen: app.getScreen()) {
    logger.log("--------------------------------------------------");
    logger.log("INFO Creating Files for screen: " + screen.getName() + " ...");

    // Create an activity file for the current screen
    activity = new ActivityHashmapper(app.getPackage(), screen, logger);
    createFileOutputStream(screen, ACTIVITY);
    processTemplating("activity.ftl", activity.getVars(), output);

    // Create a layout file for the current screen
    layout = new LayoutHashmapper(screen, logger);
    createFileOutputStream(screen, LAYOUT);
    processTemplating("layout_base.ftl", layout.getVars(), output);
}
```

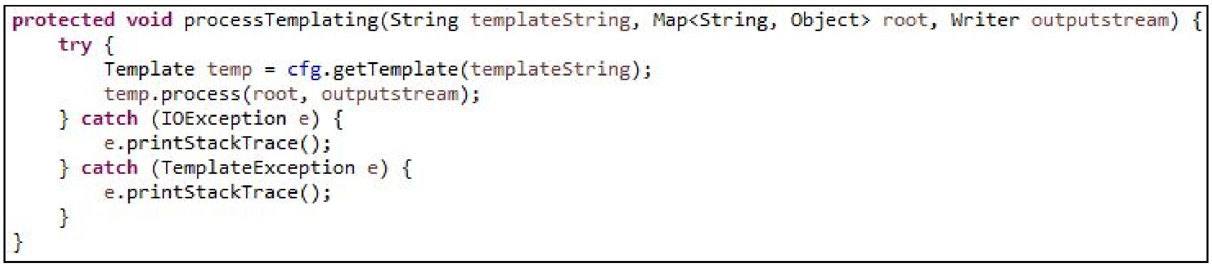
org.accapto.tool.templating.Hashmapper

org.accapto.tool.templating.Hashmapper is an abstract class that acts as a base class for files that have to be created. In general, this class generates a hashmap for the FreeMarker template engine. Therefore, subclasses return such a hashmap with FreeMarker variables. The keys in the generated hashmap are the placeholders from a specific FreeMarker template and the values are the Strings that will be put into the template instead of the placeholder.

The constructor of Hashmapper has a Logger argument and consists of a new Hashmap called ***vars***. Moreover, a standard FreeMarker configuration is generated here.

Methods

Hashmapper has two public abstract void methods called ***generateValues()*** and ***fillVars()***, which have to be implemented in the subclasses. The first one has the purpose to generate the values that will be put into the FreeMarker template, whereas the latter one is supposed to fill the hashmap with the appropriate keys for a FreeMarker template and the generated values.
The third method of this class is ***processTemplating(...)***:

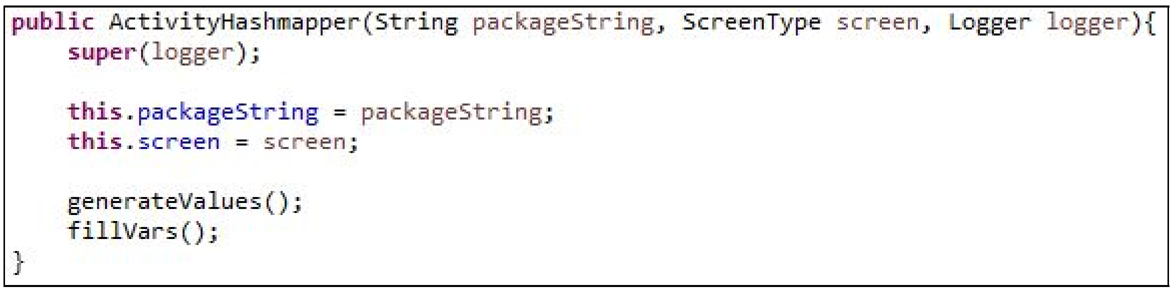


```
protected void processTemplating(String templateString, Map<String, Object> root, Writer outputstream) {
    try {
        Template temp = cfg.getTemplate(templateString);
        temp.process(root, outputstream);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (TemplateException e) {
        e.printStackTrace();
    }
}
```

This method starts the FreeMarker template engine for a specified template and templating model. The output is then written into the specified outputStream. This method is needed since some files will contain parts that are created using Freemarker templates too.

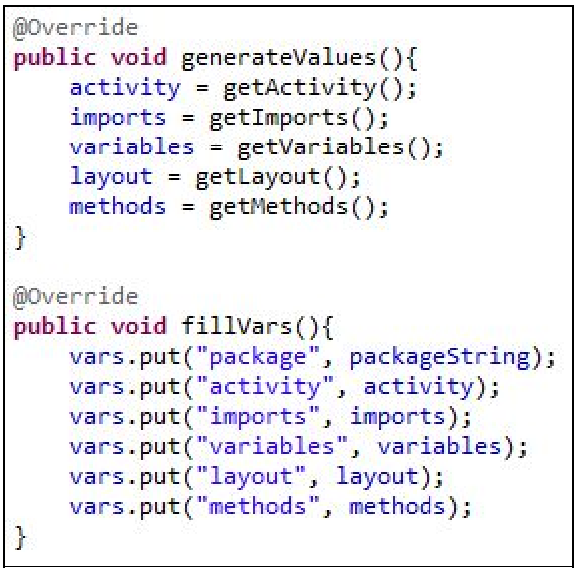
org.accapto.tool.templating.ActivityHashmapper

The **ActivityHashmapper** extends the abstract class **Hashmapper** and creates the hashmap (***vars***) for an activity file. The values that are needed for the template are the following Strings: ***packageString***, ***activity***, ***imports***, ***variables***, ***layout*** and ***methods***. Moreover, the ScreenType ***screen*** is needed for the model of the current screen.



```
public ActivityHashmapper(String packageString, ScreenType screen, Logger logger){
    super(logger);

    this.packageString = packageString;
    this.screen = screen;

    generateValues();
    fillVars();
}
```

In the constructor the methods ***generateValues()*** and ***fillVars()*** are called. The first one generates the values that will be put into the FreeMarker template (in this case the values ***activity***, ***imports***, ***variables***, ***layout*** and ***methods***), whereas the latter method fills the template with the appropriate keys for the FreeMarker template and the generated values.

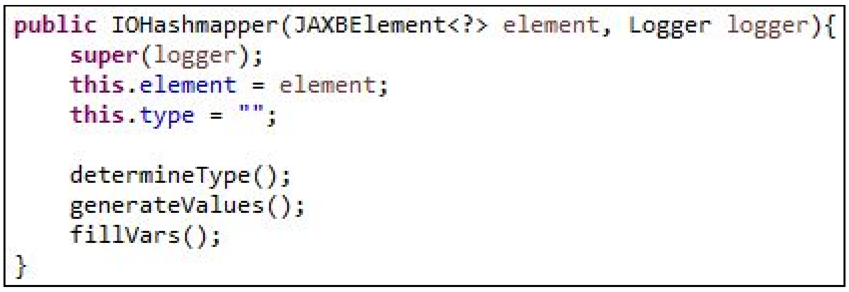


```
@Override
public void generateValues(){
    activity = getActivity();
    imports = getImports();
    variables = getVariables();
    layout = getLayout();
    methods = getMethods();
}

@Override
public void fillVars(){
    vars.put("package", packageString);
    vars.put("activity", activity);
    vars.put("imports", imports);
    vars.put("variables", variables);
    vars.put("layout", layout);
    vars.put("methods", methods);
}
```

For now, methods like ***getImports()*** and ***getVariables()*** return an empty string, but if they are needed in the future, they can be modified. At this point in time, the method ***addImports()*** adds the imports for *android.view.**View*** and *android.content.**Intent*** that are necessary for a transition. This method is called in ***getMethods()***, where a loop through all screen objects takes place if screen objects exist. Afterwards, a method stub for all <action> objects is created as well as methods with intents for <transition> objects. Eventually, the String ***temp*** gets returned for ***getMethods()***.

org.accapto.tool.templating.IOHashmapper

The **IOHashmapper** extends the **Hashmapper** and creates the hashmap for input, output, action and transition types. The values that are needed for the template are the following Strings: ***name***, ***nameNoSpace***, ***description***, ***function*** and ***transition***. Moreover, the JAXBElement<?> ***element*** is needed from the input XML file and the String ***type*** defines the type of the element, which can be an input, an output, an action or a transition.

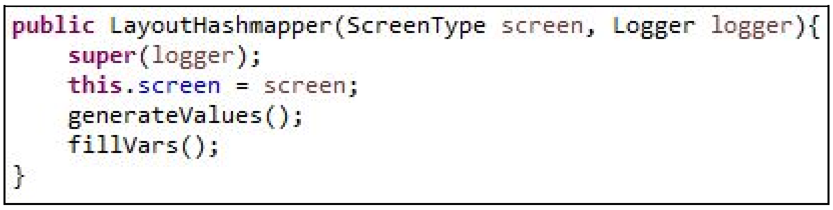


```
public IOHashmapper(JAXBElement<?> element, Logger logger){
    super(logger);
    this.element = element;
    this.type = "";

    determineType();
    generateValues();
    fillVars();
}
```

The methods ***generateValues()*** and ***fillVars()*** are called in the constructor (for more details about those, take a look at the other **Hashmapper** classes) and for a IOHashmapper instance a JAXBElement<?> ***element*** and a Logger ***logger*** are passed as arguments. Before the above mentioned methods are called, ***determineType()*** is executed to determine the type of the element (input, output, action or transition) and save it in the appropriate variable to avoid checking it again and again.
Some types do not require all variables for their template; however all types are treated the same for convenience (key-value pairs in the hashmap that are not needed by freemarker will not cause exceptions).

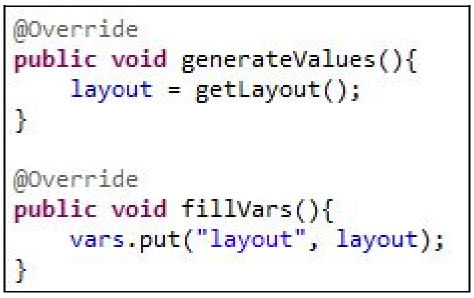
org.accapto.tool.templating.LayoutHashmapper

The **LayoutHashmapper** extends the abstract class **Hashmapper** and creates the hashmap for a layout file. The value that is needed for the template is the String ***layout***. Moreover, the ScreenType ***screen*** is needed for the model of the current screen.



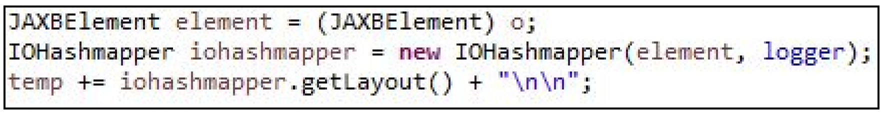
```
public LayoutHashmapper(ScreenType screen, Logger logger){
    super(logger);
    this.screen = screen;
    generateValues();
    fillVars();
}
```

In the constructor the methods ***generateValues()*** and ***fillVars()*** are called. The first one generates the value that will be put into the FreeMarker template (in this case the value ***layout***), whereas the latter method fills the template with the appropriate key for the FreeMarker template and the generated value.



```
@Override
public void generateValues(){
    layout = getLayout();
}

@Override
public void fillVars(){
    vars.put("layout", layout);
}
```

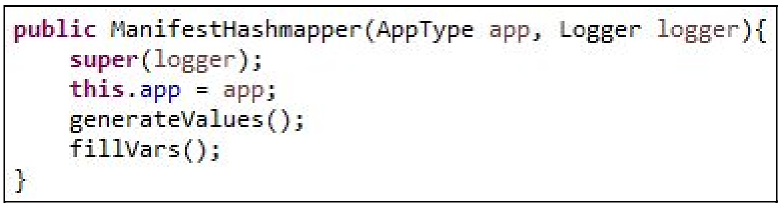
The method ***getLayout()*** returns the String ***temp***, which is created via a loop through all screen objects if screen objects exist. In this loop, the layout for <input>, <action> and <transition> objects is generated. This is done by creating Strings from templates via a new instance of the class **IOHashmapper**.



```
JAXBElement element = (JAXBElement) o;
IOHashmapper iohashmapper = new IOHashmapper(element, logger);
temp += iohashmapper.getLayout() + "\n\n";
```

org.accapto.tool.templating.ManifestHashmapper

The **ManifestHashmapper** extends the abstract class **Hashmapper** and creates the hashmap for the manifest file. The values that are needed for the template are the following Strings: ***packageString***, ***activities*** and ***permissions***. Moreover, the AppType ***app*** is needed for the model of the app and the String ***intent***, because an intent-filter is added to the first screen defined in the input file (app won't work without intent-filter, because it is confused and doesn't know which screen is the "start" screen).



```
public ManifestHashmapper(AppType app, Logger logger){
    super(logger);
    this.app = app;
    generateValues();
    fillVars();
}
```

The methods ***generateValues()*** and ***fillVars()*** are called in the constructor (for more details about those, take a look at the other **Hashmapper** classes) and for a ManifestHashmapper instance an AppType ***app*** and a Logger ***logger*** are passed as arguments.

Methods

Another method in this class is ***getActivities()***, which is called in ***generateValues()*** and generates the ***activities*** value that will be put into the FreeMarker template, but first the activities must be fetched which happens in ***getActivities()****.* There, the intent-filter is added to the first screen. Afterwards, a loop over the screens is executed and ***processTemplating()*** method from the superclass is called to start the FreeMarker template engine for the specified template "manifest_activity.ftl" and the specified templating model *temp*. The output is then sent to the writer and the ***getActivities()*** method returns the outputStream as a String.