How To
Hammurapi

Version Log:

Date	Version	Status	Author	Remarks
26-July-04	1.0	Release candidate	Pavel	Covers version 3.0
18-July-04	0.9	Draft	Pavel	Reflects changes in InspectorContext and Waiver API's
05-July-04	0.8	Draft	Pavel	Filtering, detailed description of inspectors.xml
17-June-04	0.7	Draft	Pavel	Chapter 5 describes new command-line interface. Chapter 6 modified.
11-June-04	0.6	Draft	Pavel	Chapter 17
25-May 04	0.5	Draft	Pavel	Chapters 14, 16
25-May 04	0.4	Draft	Johannes	Amits Remarks & add. Chapter 11,13,15
17-May-04	0.3	Draft	Pavel	Added chapters 8,9,10
12-May 04	0.2	Draft	Johannes	Added main() chapter
8-May 04	0.1	Draft	Janos	

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

Table Of Content

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

1 Introduction

Hammurapi is a code review tool. It scans Java source files and inspects them to adherence with coding standards. Hammurapi produces reports in HTML or XML. It can be executed either as standalone application or as Ant task.

2 System requirements

Hammurapi requires Java 1.3.x or Java 1.4.x. By default lib directory contains jar files for Java 1.4. If you intend to run Hammurapi on Java 1.3 then replace lib/pvcommons-…-1.4.jar with lib/pvcommons-…-1.3.jar in Hammurapi classpath.
Hammurapi can review Java 1.4 or earlier sources. Java 1.4 grammar is incompatible with Java 1.3 grammar because **assert** is a reserved word in Java 1.4. So if you have to review Java 1.3 sources which contain methods **assert(…)** you need to replace lib/Jsel-…-1.4.jar with lib-1.3/Jsel-…-1.3.jar in Hammurapi classpath.

3 Concepts

Hammurapi works with a **Repository** of java files. **Repository** is a file, directory or a group of files/directories. Hammurapi parses files in **Repository** and then navigates **Inspectors** through the parsed files.
Inspector is a java class, which inspects a piece of java source code. **Inspectors** report their findings to Hammurapi by creating **Violations**, issuing **Warnings**, creating **Annotations**, or gathering **Metrics.**
Hammurapi accumulates **Inspector's** findings and creates a report. It is possible to provide **Waivers** to Hammurapi to waive some particular **Violations** or all **Violations** found by particular **Inspector**.
Inspectors and **Violations** generated by them have **Severity,** which is a positive number. **Severity 1** is considered the highest severity. **Severity** 5 and after are considered as information messages and excluded from **Sigma** and **DPMO** calculations.
Inspectors are configured using **Inspector descriptors,** which are grouped into **Inspector sets**. **Inspector sets** can be loaded from files or URL's. Multiple **Inspector sets** can be loaded for one review. **Inspectors** can be incrementally configured by multiple **Inspector descriptors** with the same name. For example one **Inspector descriptor** can set **Inspector** class name and severity, another, loaded after the first one can override severity and also provide description.
Inspector sets support inheritance – one **Inspector set** can be based on another one. In this way you can have base **Inspector set** and specialized **Inspector sets** for different application types. E.g. one **Inspector set** for EJB Applications and another **Inspector set** for Swing applications.
Waivers are grouped in **Waiver Sets,** which can be loaded from files and URL's. Multiple **Waiver Sets** can be loaded for one review.
Waiver can be applied to one instance of **Violation** or to all **Violations** reported by a particular **Inspector. Waiver** can have an expiration date. One **Inspector** can waive **Violations** reported by another **Inspector.** E.g. you have an general policy that prohibiting hiding inherited instance variables. There is an **Inspector** checking adherence to this policy. At the same time you have another policy that every class shall have its own logger with predefined name *logger.* And you have an **Inspector** checking for this. This policy conflicts with "Prohibit hiding inherited instance variables" policy. It is

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

possible to configure the second **Inspector** to waive violation reported by the first **Inspector.**
Review results are reported to **Listeners**. Hammurapi comes with an **Output Listener** which generates reports in HTML or XML. You can write your own listeners which can provide custom reporting. You can also turn this **Listener** off if you don't like the default reporting functionality.
If you run Hammurapi as part of build process you can prevent low quality code to be deployed by specifying code quality thresholds. Hammurapi will fail build if source code does not meet specified standards.

4 Installation

For running Hammurapi you need Hammurapi distribution. If you plan to run Hammurapi as Ant task then you need to install Ant as well.
It is possible to run Hammurapi as a standalone Java application, but it is much more configurable as an Ant task.
Ant can be downloaded from the website http://ant.apache.org.

4.1.1 Downloading Hammurapi distribution

Hammurapi can be downloaded from the website www.hammurapi.org. There are binary and source distributions The binary distribution contains all the jars required to run Hammurapi.

5 Running Hammurapi from command line

Hammurapi can be launched from the command line using either hammurapi script on Linux/Unix or hammurapi.bat batch file on Windows. You might want to set HAMMURAPI_HOME environment variable before you run Hammurapi, though it is not required. It is also recommended to add $HAMMURAPI_HOME/bin to path to be able invoke Hammurapi from any directory.
Some inspectors require particular classes in Hammurapi classpath. For example EJB inspectors need SessionBean interface in classpath. To avoid adding j2ee.jar, log4.jar and struts.jar to Hammurapi classpath every time you run review you can add "-c <path to j2ee.jar>:<path to log4j.jar>:<path to struts.jar>" to HAMMURAPI_ARGS environment variable.
If you run hammurapi without arguments of with –h option it will output usage summary:

```
usage: Usage: hammurapi [options] <output dir> <source files/dirs>
 -D <local database>      Database name
 -R <database server>     Database server name
 -S <severityThreshold>   Severity threshold
 -T <title>               Report title
 -U <waivers url>         Waivers URL
 -W <waivers file>        Waivers File
 -c <classpath>           ClassPath
 -d <dpmoThreshold>       DPMO Threshold
 -e                       Do not load embedded inspectors
 -g                       Debug
 -h                       Print this message
 -i <inspectorsFile>      Inspectors file
 -j <user name>           Database user
 -l <class name>          Review listener
```

```
-o                          Suppress output
-p <password>               Database password
-r                          Wrap Jsel model
-s <sigmaThreshold>         Sigma threshold
-t <debug type>             Jsel type to debug
-u <inspectorsURL>          Inspectors URL
-v                          Verbose
-w <waiverStubs>            Where to output waiver stubs
-x                          Output XML
```

5.1 Options

- D – Local database name. E.g. `C:\MyProject\HammurapiDB`
- R – Database server name. E.g. `localhost`.
- S – Severity threshold. Integer number. If review contains violations with severity equal or less than the threshold then Hammurapi exit code will be 2.
- T – Report title. E.g. `MyProject`
- U – Waivers URL. URL where to read waivers from. Can be specified zero, one or more times and mixed with –W.
- W – Waivers file. File where to read waivers from. Can be specified zero, one or more times and mixed with –U.
- c – Classpath. Can be specified more than once.
- d – DPMO threshold. Integer number. If review DPMO is higher than the threshold then Hammurapi will return 2.
- e – Do not load embedded inspectors. Use this option if you use custom inspectors.xml.
- g – Sets logging level to DEBUG.
- h – Prints help message and exits.
- i – File to load inspectors from. Can be specified multiple times and combined with –u.
- j – User name for database server.
- l – Review listener class name. The class must implement org.hammurapi.Listener and have public no argument constructor. Can be specified more than once.
- o – Suppress output. If you provide this option than no output will be generated. This option is useful in combination with –l option.
- p – Database password (for server).
- r – Wrap Jsel model. Use this if you have big source bases and inspectors, which keep references to Jsel model elements. Rule of thumb – try to use it if you get OutOfMemoryError.
- s – Sigma threshold. Double number. If review sigma is below than the threshold then Hammurapi exit code will be 2.
- t – Debug type. Class name of Jsel type to debug. E.g. com.pavelvlasov.jsel.VariableDefinition. Hammurapi will output detailed information about inspectors inspecting VariableDefinition.
- u – URL to load inspectors from. Can be specified multiple times and combined with –i.

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

- v – Sets logging level to VERBOSE.
- w – File to output waiver stubs to.
- x – Tell Hammurapi to output report in XML format.

6 Hammurapi in Eclipse via Main

You can start Hammurapi from Eclipse by creating a Run/Debug configuration for org.hammurapi.HammurapiTask.main() method and providing argument as described in the previous section.
Add a new Java App in the Eclipse Launcher.

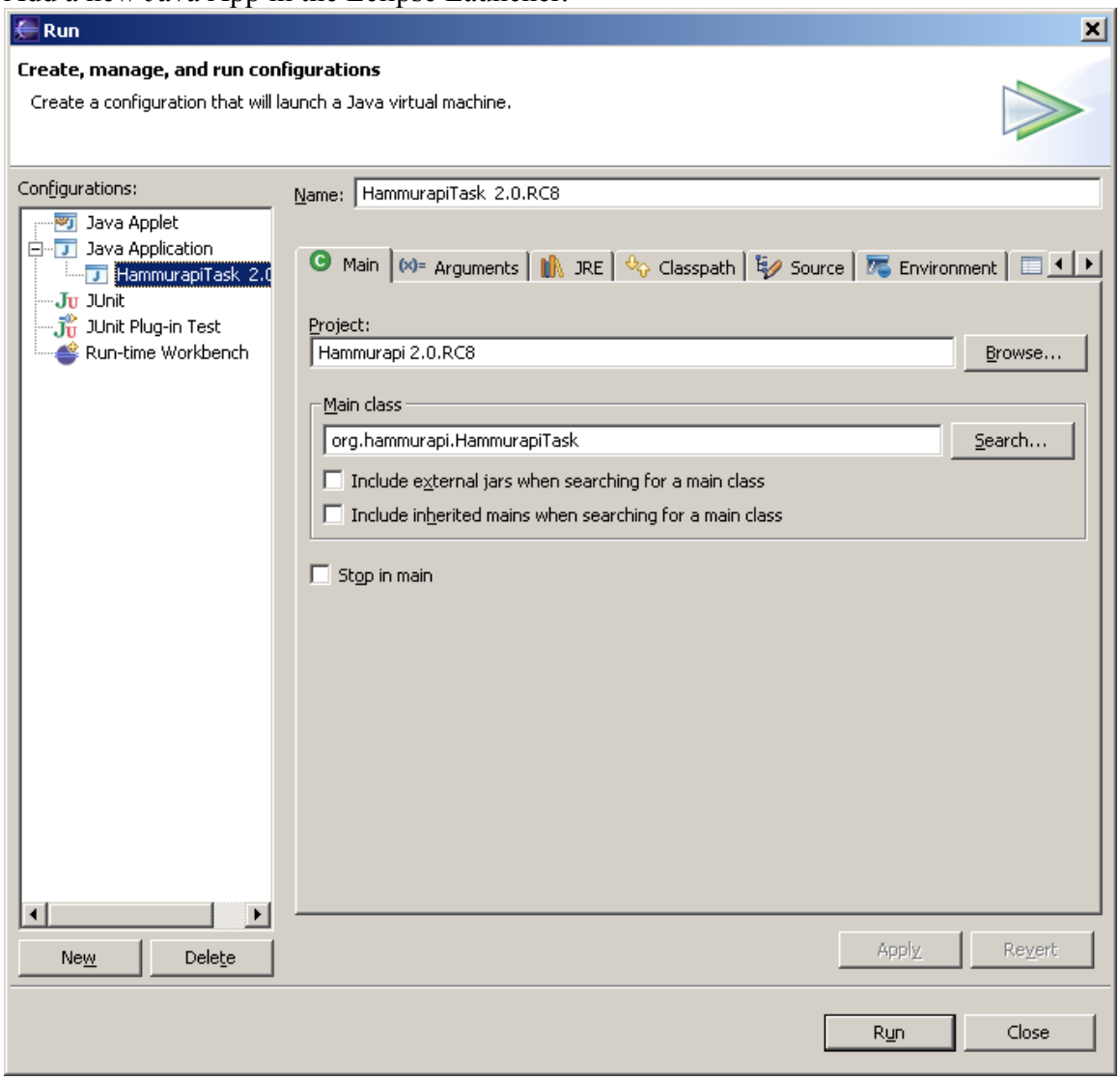

Figure 1

It is recommended to specify the Heap size with the VM parameters e.g.
-Xms256m –Xmx512m
on a 1GB box. You can play around with this parameters, but its recommended not to exceed the physical memory and also let he OS a small portion.

If you want to review an existing Eclipse project and don't want to mess with –c option then you need to make that project visible to Hammurapi. Just open the Properties on your Hammrapi project and mark this project, which shall be reviewed.
If you miss this step, Hammurapi may give you tons of "Class not Found Warnings".

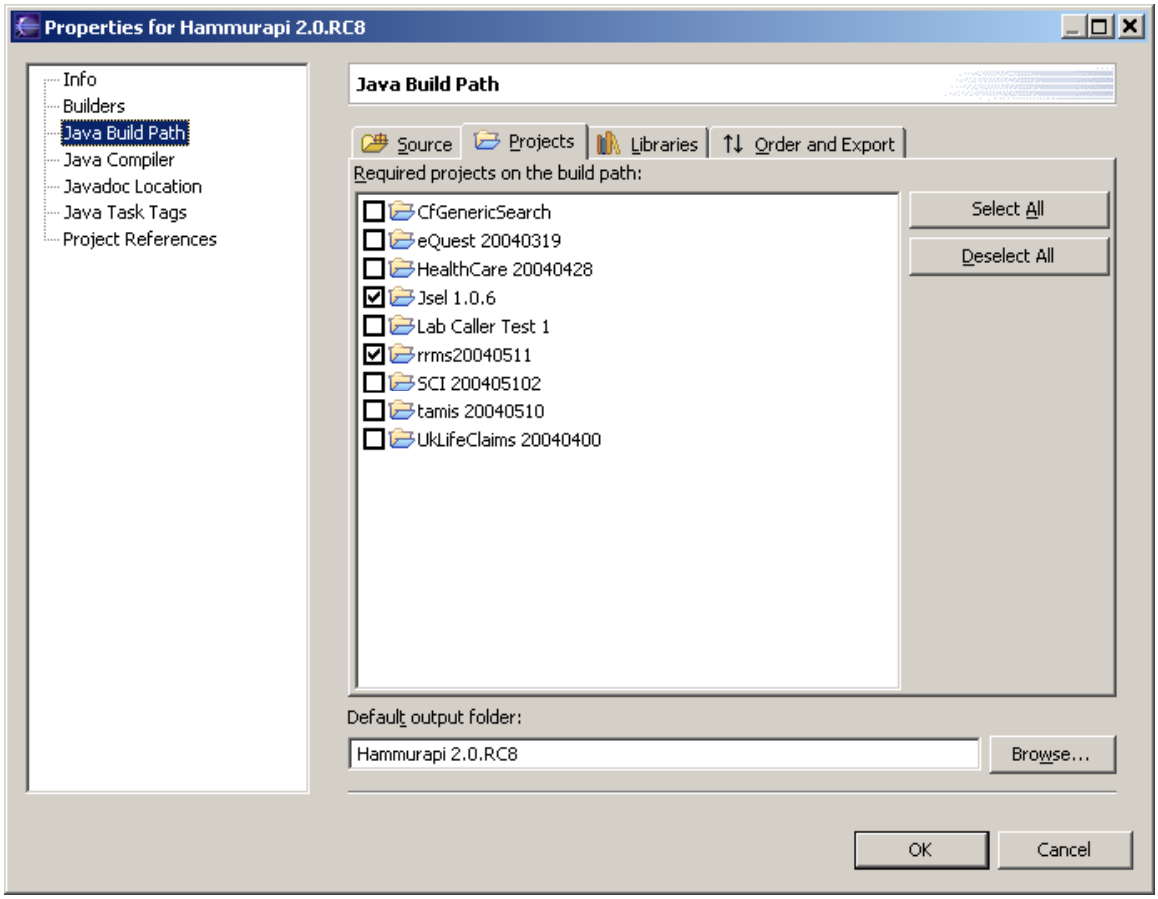

Figure 2

Please set the Project Home Directory to your Hammurapi Installation or to you project source code directory.

7 Hammurapi in Eclipse as an Ant target

7.1 Configuring a Hammurapi Ant task

Hammurapi Ant configuration is appreciated if you want to do a code review as part of your build process. This is a great advantage compared to GUI centred code review tools and fulfill the need of companies with an automated build & deployment process. Furthermore, the memory footprint of Eclipse can have a negative impact on your code review on a box with small memory.

7.1.1 The ant build.xml file

This is an example, where I've downloaded the source binary of Hammurapi. The build.xml doesn't contain any hard coded folder. Everything is configured in the file build.properties.
With this configuration it is easy to change the inspected project very fast. I have a separate build.properties file for every inspected project. To change the inspected project the following items have to be modified:

- ${hammurapi.inspect.project.src}
- ${hammurapi.inspect.project.bin}
- {hammurapi.inspect.project.lib}

```
<project name="jTtaste1" basedir="." default="test">

<property file="build.properties"/>

<taskdef
  name="hammurapi"
  classname="org.hammurapi.HammurapiTask"
>
  <!-- class path wich for running Hammurapi -->
  <classpath>
    <!-- The jars are in this folder which are necessary to run Hammurapi -->
    <fileset dir="${hammurapi.lib}" includes="*.jar"/>
    <!-- jars which are necessary for the built in rules -->
    <fileset dir="${hammurapi.extra.lib}" includes="*.jar"/>
    <!-- jars which are necessary for the inspected project -->
    <fileset dir="${hammurapi.inspect.project.lib}" includes="*.jar"/>
    <!-- zip files (classes12.zip of Oracle) which are necessary for the inspected
project -->
    <fileset dir="${hammurapi.inspect.project.lib}" includes="*.zip"/>
  </classpath>
</taskdef>

<target name="test">
  <hammurapi>
    <!-- The source folder of the inspected project -->
    <src dir="${hammurapi.inspect.project.src}"/>
    <classpath>
      <!-- jars which are necessary for the built in rules -->
      <fileset dir="${hammurapi.extra.lib}" includes="*.jar"/>
      <!-- The folder which contains the compiled classes of the inspected project -->
      <path location="${hammurapi.inspect.project.bin}"/>
      <!-- jars which are necessary for the inspected project -->
      <fileset dir="${hammurapi.inspect.project.lib}" includes="*.jar"/>
      <!-- zip files (classes12.zip of Oracle) which are necessary for the inspected
project -->
      <fileset dir="${hammurapi.inspect.project.lib}" includes="*.zip"/>
    </classpath>
    <!-- The output folder where Hammurapi saves the inspection result -->
    <output dir="${hammurapi.inspect.project.review}"/>
  </hammurapi>
```

```
</target>
</project>
```

Important items are:

- The folder ${hammurapi.extra.lib}" conatins the jars, which are necessary to use the built in rules. There are rules for javax.ejb.EnterpriseBean's and so on… These classes are not part of Hammurapi, but are mandatory for using the built in rules.
- The folder ${hammurapi.inspect.project.lib}" contains the jar files, which are mandatory for inspecting the target project. It is dependent of that project, so cannot be part of Hammurapi.

More possible Ant configurations can be read at the website www.hammurapi.org.
Hammurapi has been tested with Ant 1.5.2, but I was able to use it with the latest version of Ant.

7.1.2 Memory usage

It is recommended to set a high memory usage for the JRE, which runs Ant and as an Ant task Hammurapi. The easiest way to do it is the setting in the file ant.bat, which can be found in the bin folder of the Ant distribution.

7.2 Basic configuration of Eclipse

I'm using Eclipse 2.1.2 with the Ant plug-in for Ant 1.5.3.

7.3 Running Ant inside Eclipse

After right clicking on the item build.xml in the View Package Explorer, the next popup appears.

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

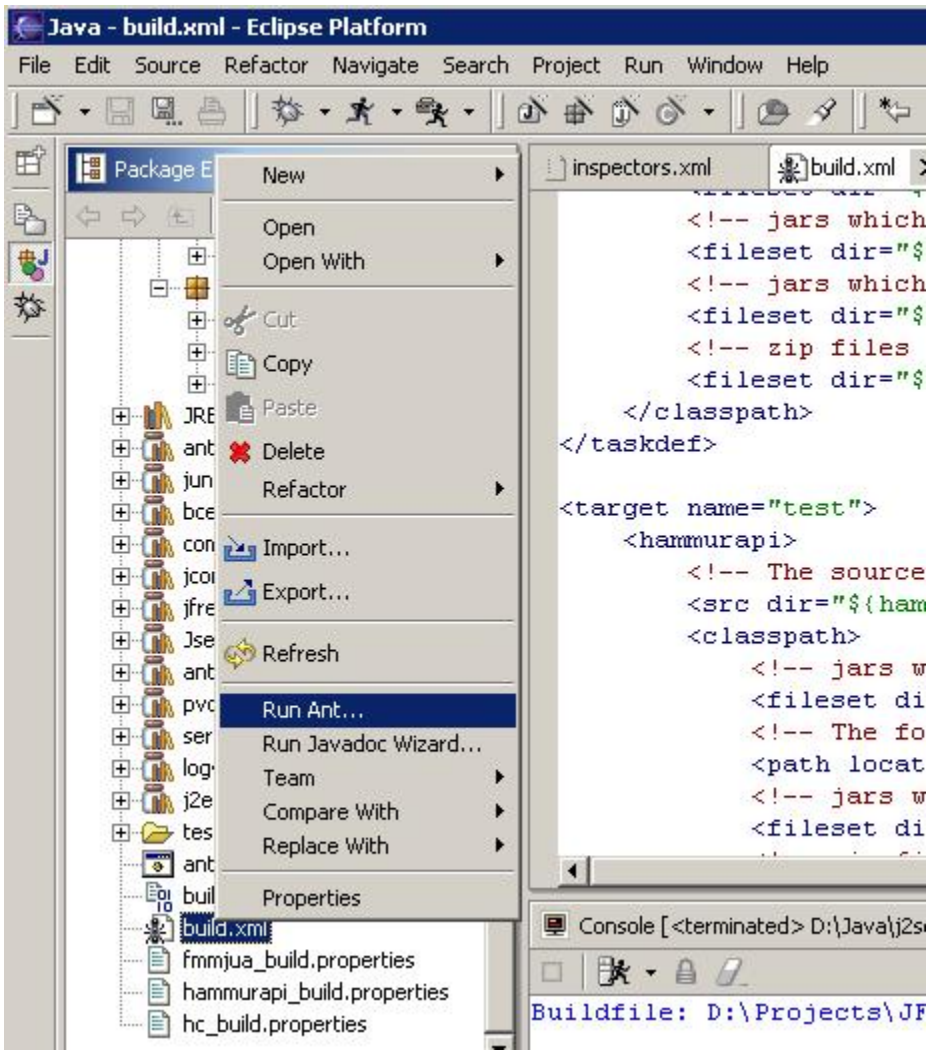

Figure 3

Selecting Run Ant…, Eclipse shows the External Tools launch dialog. In this dialog, it is possible to select Ant target(s) and to let it(them) run. These steps create a persisted launch configuration. The newly created configuration will appear in the launch history under **Run > External Tools** and will be available in the launch configuration dialog which is opened by clicking **Run > External Tools > External Tools...**. You can find more information in the Eclipse help under "Running Ant build files".

7.4 Running Ant as a normal Java Application

In the Eclipse help you can read detailed information about this topic under "Launching a Java program". In order to launch Ant, you have to configure properly your launch item. This example has been made with the Ant launch plug in from aloba ag, which is reachable under the URL
http://www.aloba.ch/produkte/veroeffentlichungen/antviewdebug/index.html.
First select the menu Navigate/Run… ! In the Run dialog make a new configuration item as an Application and fill out the main class!

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

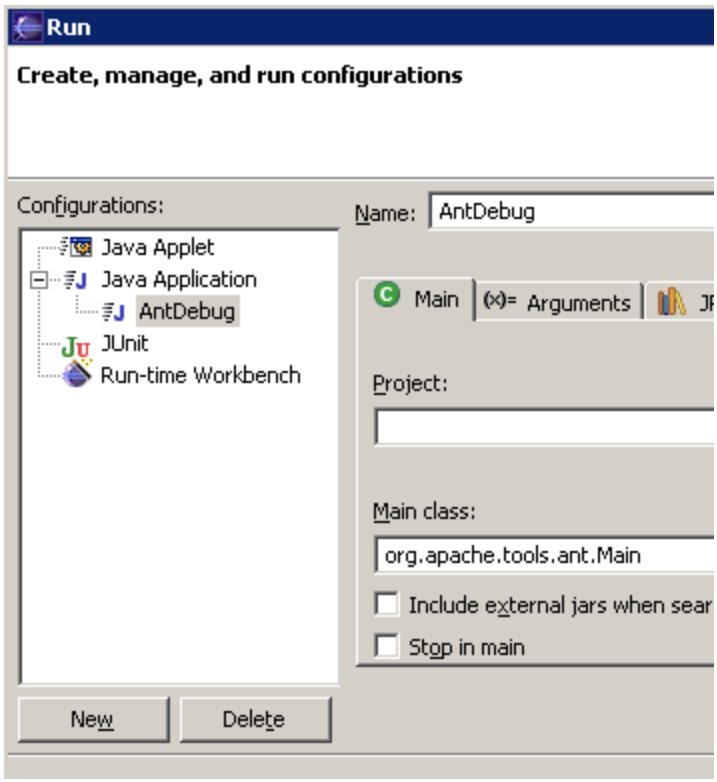

Figure 4

Select the (x) Arguments tab and fill the Program arguments field with the value -build file <YOUR_PATH>\build.xml. This item should point to your build.xml file.

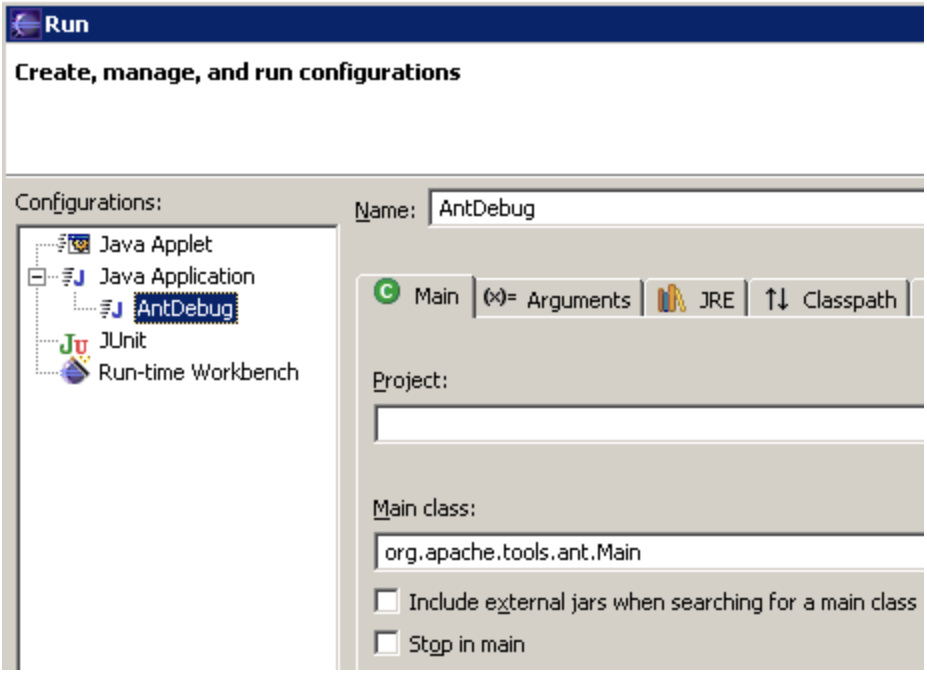

Figure 5

On the tab Class path you have to set the following jars:

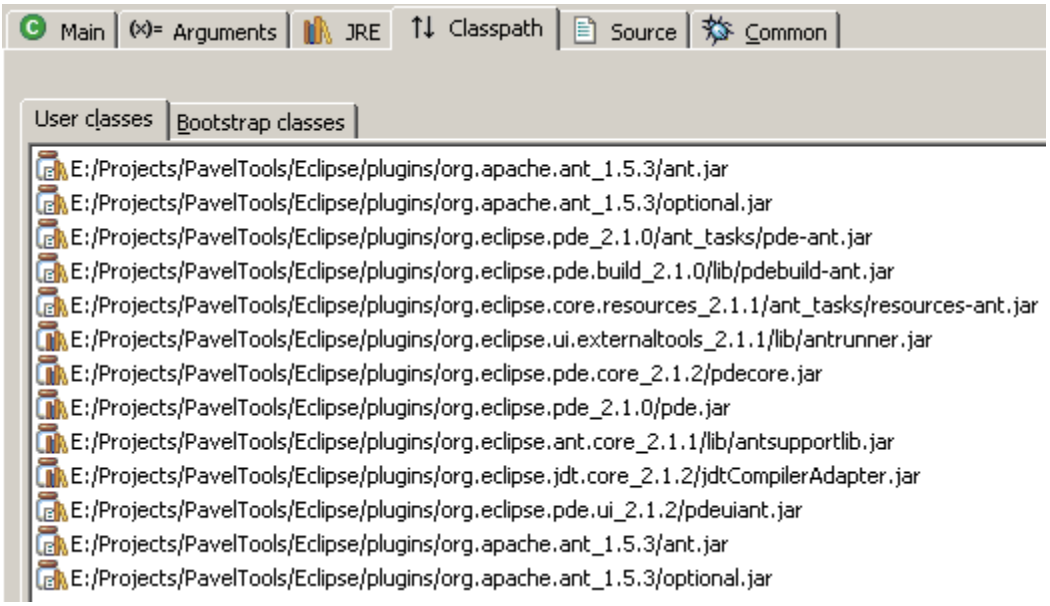

Figure 6

These jars either belong to the Ant plug in of Eclipse, or to the Eclipse plug in framework.
On the tab Source You have to set the following items:

- **HammurapiSrc**
- <JAVA_HOME>/lib/tools. jar
- <ECLIPSE_HOME>/plugins/org.apache.ant_1.5.3/ant.jar
- <ECLIPSE_HOME>/plugins/org.apache.ant_1.5.3/optional.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.pde_2.1.0/ant_tasks/pde-ant.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.pde.build_2.1.0/lib/pdebuild-ant.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.core.resources_2.1.1/ant_tasks/resources -ant.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.ant.core_2.1.1/lib/antsupportlib.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.ui.externaltools_2.1.1/lib/antrunner.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.pde.core_2.1.2/pdecore.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.pde_2.1.0/pde.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.jdt.core_2.1.2/jdtCompilerAdapter.jar
- <ECLIPSE_HOME>/plugins/org.eclipse.pde.ui_2.1.2/pdeuiant.jar

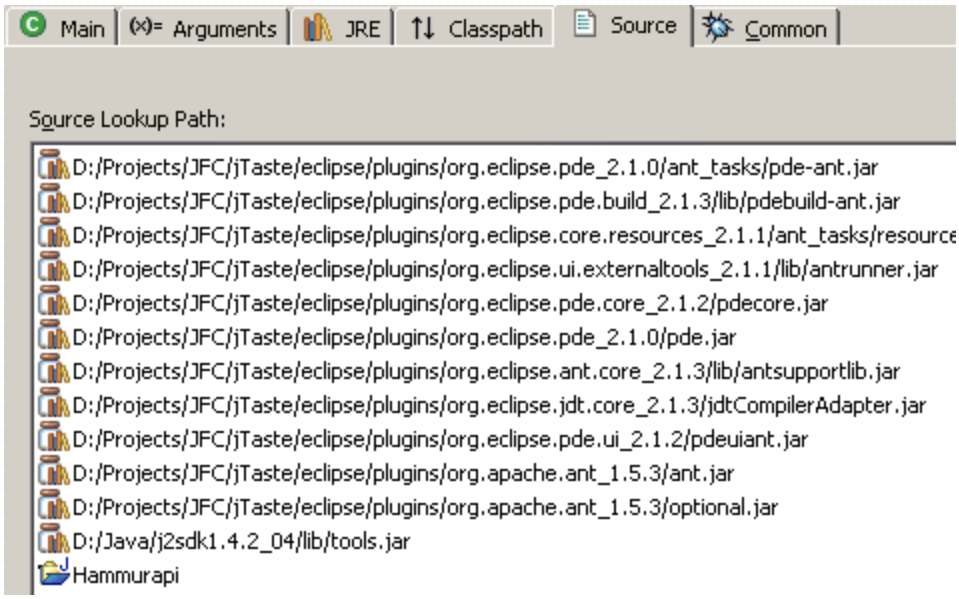

Figure 7

One of the most important item in this list is HammurapiSrc, which points to the source files of our Hammurapi distribution. It is important, if we want to debug our Hammurapi Ant project. The Ant plug in of aloba ag doesn't add this item to the Source Lookup Path since it touches our project.
We can add this item to the source list when we uncheck "Use default source lookup path" and click on the button "Advanced…". You can read in the Eclipse Help under "Creating a Java application launch configuration" about more details.
After creating this configuration it is possible to launch our Ant project with the Run, or with the Debug item on the Toolbar of Eclipse.

8 Development process with Hammurapi

We recommend the following way to embed Hammurapi in your development process:

- There are two roles – Architect and Developers.
- Architect prepares inspector set or hierarchy of inspector sets and configures Hammurapi to either be part of build process, in which case Hammurapi can fail builds on high severity or low PAI, or run Hammurapi separately, say, once a week.
- Architect also controls waivers file(s).
- For legacy codebase Architect can use "Comply-on-touch" policy (see page 28).
- Development team performs coding and then runs Hammurapi agains their code.
- Then developers analyze Hammurapi findings and either fix violations or request waivers from Architect or request architect's help.
- Architect review Developer's requests and either gives waivers or suggests how to fix violations.
- Architect may manually review most interesting pieces of code and based on her/his findings introduce new inspectors and/or annotators.

9 Waivers

Waivers are the way to tell Hammurapi that some findings aren't actually violations. For example sometimes you need to have empty catch block as in the example below:

```
integer=defaultValue;
if (string!=null) {
  try {
    integer=Integer.parseInt(string);
  } catch (NumberFormatException e) {
    // we don't want to do anything here
  }
}
```

Waived violations will appear in the report as shown below:

Waived violations

#	Line	Column	Name	Severity	Description	Waiver reason	Waiver expires
1	57	19	ER-002	1	Empty catch block	To demonstrate waivers in action	2004/05/18

9.1 Defining a waiver

To waive a violation for the example above you need to do the following:

1. Define ER-002 as waivable. You can do it by modifying inspectors.xml

```
<inspector-descriptor>
  <name>ER-002</name>
  <enabled>yes</enabled>
  <severity>1</severity>
  <inspector type="org.hammurapi.inspectors.EmptyCatchBlockRule"/>
  <description>Empty catch block</description>
  <waivable>yes</waivable>
  <waive-case>
    In some situations exception is excepted and shall be ignored.
    Example:
      int i=0;
      if (str!=null) {
        try {
          i=Integer.parseInt("hello");
        } catch (NumberFormatException e) {}
      }
  </waive-case>
…
</inspector-descriptor>
```

   You can also add waive-case element describing situations in which this exception can be waived. Another way to declare inspector as waivable is by adding nested element to Hammurapi task:

```
<hammurapi waiverStubs="waiver_stubs.xml" title="Test cases">
  <inspector name ="ER-002" waivable = "yes">
  …
</hammurapi>
```

2. Set `waiverStubs` attribute in hammurapi task: waiverStubs="waiver_stubs.xml
3. Run Hammurapi. It will create waiver_stubs.xml file with entries like below.

```
<waiver>
    <inspector-name>ER-002</inspector-name>

<signature>org/hammurapi/inspectors/testcases/violations/EmptyCatchBlockRuleViolat
ionTestCase.java:at[EmptyCatchBlockRuleViolationTestCase]:ao[getFirstByte(java.lan
g.String)]:eo[1]:es[java.io.IOException]</signature>
     <reason>This exception is ignored for testing purposes.</reason>
     <expiration-date>2004/05/15</expiration-date>
</waiver>
```

   Rename or copy it to waivers.xml and leave <waiver> elements only for violations which you are going to waive. If you want to waive all violations for a

particular inspector then delete <signature> element. If you don't want waiver to expire then delete <expiration-date> element. One waiver element can contain multiple <signature> elements, but in waiver stubs every signature is placed in individual <waiver> element.

4. Add <waivers> element to the task

```
<hammurapi waiverStubs="waiver_stubs.xml" title="Test cases">
  <waivers file="waivers.xml"/>
  <inspector name ="ER-002" waivable = "yes">
  …
</hammurapi>
```

And run it again. Violations listed in waivers.xml will be waived.

9.2 Format of <waiver> element

<waiver> element supports the following nested elements:

- **inspector-name –** Inspector name.
- **signature –** Code signature. There can be zero, one or more signature elements. If number of signature elements is zero then waiver applies to all inspector's reported violations.
- **include-element –** package, type or operation to include. ***** matches anything, *package.****** matches *package* and its subpackages. Packages are matched literally, types are matched based on isKindOf(), including operation types as well. If operation doesn't contain type name then any type will match., **?** matches any type in operation parameters. `<init>` shall be used for constructor names.
- **exclude-element –** package, type or operation to exclude. Same format as for **include-element.**
- **include-file** – File name or file name pattern to include. * states for zero or more characters. ? states for zero or one character.
- **exclude-file –** File name or file name pattern to exclude. Same format as for **include-file.**
- **reason –** Reasoning behind waiving decision.
- **expiration-date –** Waiver expiration date. If this element is missing then waiver never expires.

Includes/excludes are applied in the reverse definition order, by layers. This means that definitions of operations are applied first, then definitions of types, then of files, and after that of packages. It is a good practice not to mix different layers (package, file, type, operation) in one waiver.

Examples of include/exclude:

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

- `<include-element>org.hammurapi</include-element>` - includes package org.hammurapi
- `<exclude-element>org.hammurapi.*</exclude-element>` - excludes package org.hammurapi and its subpackages
- `<exclude-element>org.hammurapi.InspectorDescriptor</exlude-element>` - excludes interface org.hammurapi.InspectorDescriptor and its implementations
- `<include-element>org.hammurapi.InspectorDescriptor.getMessage(java.lang.String)</include-element>` - includes method interface org.hammurapi.InspectorDescriptor.getMessage(java.lang.String) and its implementations
- `<exclude-element>toString()</exclude-element>` - excludes toString() in all classes/interfaces
- `<exclude-element>org.somepackage.SomeClass.<init>(?, java.util.List)</exclude-element>` - excludes constructors which take two parameters, the first one of any type and the second one of type `java.util.List` or its implementations in `org.somepackage.SomeClass` and its subclasses
- `<include-file>*.bpf</include-file>` - includes .bpf files

9.3 Scenarios

There are several scenarios for using waivers:

- There is a conscious decision to violate some rule. Developer communicates that decision to architect. If Architect agrees with Developer's decision then (s)he adds a waiver without expiration date to waivers file.
- There is an organizational policy in place, which prohibits deployment to production environment code with Severity 1 violations. There is also code, which contains Severity 1 violations, but due to time constraints these violations cannot be fixed immediately. Developer requests Architect to give a temporary waiver. Architect gives a waiver with expiration date. Developer must fix the violation by that date.
- Architect introduces a new inspector or a set of inspectors. (S)he puts waiver(s) with expiration date, but without signature. If waiver signature is blank that waiver applies to all inspector findings. Developers will see waived violations in Hammurapi reports and will have to fix them by compliance date (waiver expiration date) set by Architect.
- Some classes/packages plays special role in your application and some inspectors shall not apply to them.

10 Auto waivers

Autowaivers allow one inspector waive finding of another. For example if you code complies with "**ER-049** Unify logging strategy - define individual logger for class" then it violates "**ER-075** Avoid hiding inherited instance fields". To avoid this situation ER-049 automatically waives ER-075 by calling context.waive(element, "AvoidHidingInheritedInstanceFields"). The first parameter is the element for which waiver is given. The second parameter is a logical name of the inspector which finding is being waived.

To enable autowaiving you have to add <waives> element to the descriptor of the autowaiving inspector:

```
<inspector-descriptor>
  <name>ER-049</name>
…
  <waives key="AvoidHidingInheritedInstanceFields">
    <name>ER-075</name>
    <reason>Logger is intended to hide superclass logger</reason>
  </waives>
</inspector-descriptor>
```

11 Filtering

Filtering allows one inspector filter another, which means stop visit() method of inspector being filtered from being invoked. This concept is similar to autowaiving, but autowaiving is more precise and elaborate mechanism. Autowaived violations appear in the report in "Waived violations" section. Filtering prevents inspector from visiting the node being filtered and thus no violation is ever reported.

Filter – Inspector is a many-to-many relationship. One filter can filter multiple inspectors and one inspector can be filtered by multiple filters. Filter may be associated with inspector by name or by category.

Filtering is configured through <inspector-descriptor> element in inspectors file (see Chapter 0) or <inspector> element in hammurapi task (see http://www.hammurapi.org/content/doc/ant/org/hammurapi/HammurapiTask.html).

Let's consider an example. We want to filter Cyclomatic Complexity Inspectors for _jspService(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response). Here are the steps:

1. We implement FilteringInspector. This inspector can have regular visit() and leave() methods but in our case it has only approve() method.

```
public class JspServiceFilter extends BaseInspector implements
  FilteringInspector {

  public boolean approve(Method method) {
    try {
      return !("_jspService".equals(method.getName())
          && method.getParameters().size()==2
          &&
method.getEnclosingType().isKindOf("org.apache.jasper.runtime.HttpJspBas
e")
          && ((Parameter)
method.getParameters().get(0)).getTypeSpecification().isKindOf("javax.se
rvlet.http.HttpServletRequest")
          && ((Parameter)
method.getParameters().get(1)).getTypeSpecification().isKindOf("javax.se
rvlet.http.HttpServletResponse"));
    } catch (JselException e) {
      context.warn(method, "Filter failed: "+e.getMessage());
      return true;
    }
  }
}
```

We don't need to implement any of FilteringInspector methods because they are already implemented by BaseInspector.

For any argument approve() method is invoked only once regardless of number of inspectors being filtered.

2. Now filtering inspector shall be defined in inspectors.xml file:

```
<inspector-descriptor>
  <name>JSP_SERVICE_FILTER</name>
  <enabled>yes</enabled>
  <severity>5</severity>
  <inspector
type="org.hammurapi.inspectors.filters.JspServiceFilter"/>
  <description>Filters inspectors inside _jspService()
method.</description>
  <filter name="ER-011"/>
</inspector-descriptor>
```

In our case severity is not applicable because this inspector doesn't report any violations, we put 5 "just for the case".
Because filters are inspectors they can have their own visit() and leave() methods, which can actually make decision approve/disapprove and approve will just convey this decision (see LanguageElementFilter). In such a case it is important that visit() of filter should be invoked before visit() of inspector being filtered and leave() of filter should be invoked after leave() of inspector being filtered. Hammurapi ensures it automatically by ordering invocations of visit() and leave() methods. If you happen to have a loop in filtering (inspector A filters B and B filters A and both of them have visit() method) then exception will be thrown.

11.1 LanguageElementFilter

Hammurapi ships with built-in filter `org.hammurapi.inspectors.filters.LanguageElementFilter`. Initial behavior of the filter is to approve everything. This filter supports **include**, **exclude**, **include-file**, and **exclude-file** parameters. Order is significant - include/exclude defined later takes precedence over previous definitions. So including org.hammurapi.* and then excluding org.hammurapi.inspectors.* makes sense, but not vice versa.
If package excluded all types and operations in it are excluded. If type is excluded all operations in this type are excluded. Operation parameters can contain **?**, which matches any type.

<init> shall be used as operation name for constructors.

Parameter format:

- * - matches anything
- <string> - matches package, type or operation (if contains '(')
- <string>.* - matches package and its subpackages

Packages are matched literally, types are matched based on isKindOf(), including operation types as well. If operation doesn't contain type name then any type will match. **include-file** and **exclude-file** parameter values are file name patterns. **?** states for zero or one character, ***** states for zero or more characters. See org.apache.oro.text.GlobCompiler for more details.

Examples

`<parameter name="include">org.hammurapi</parameter>` - includes package org.hammurapi

`<parameter name="exclude">org.hammurapi.*</parameter>` - excludes package org.hammurapi and its subpackages

`<parameter name="exclude">org.hammurapi.InspectorDescriptor</parameter>` - excludes interface org.hammurapi.InspectorDescriptor and its implementations

`<parameter name="include">org.hammurapi.InspectorDescriptor.getMessage(java.lang.String)</parameter>` - includes method interface org.hammurapi.InspectorDescriptor.getMessage(java.lang.String) and its implementations

`<parameter name="exclude">toString()</parameter>` - excludes toString() in all classes/interfaces

`<parameter name="exclude">org.somepackage.SomeClass.<init>(?, java.util.List)</parameter>` - excludes constructors which take two parameters, the first one of any type and the second one of type java.util.List or its implementations in org.somepackage.SomeClass and its subclasses

`<parameter name="include-file">*.bpf</parameter>` - includes .bpf files.

This build file fragment shows how to filter ER-011 in _jspService() method:

```
<inspector name="_jspService() filter"
className="org.hammurapi.inspectors.filters.LanguageElementFilter">
  <filterInclude name="ER-011"/>
  <parameter name="exclude"
value="org.apache.jasper.runtime.HttpJspBase._jspService(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)"/>
</inspector>
```

12 Hammurapi Inspector Configuration

The base configuration provides 114 inspectors. If you want to change this, please unpack the JAR file and check the file *inspector.xml* in package *org.hammurapi*.
Top level element of inspectors.xml is <inspector-set> which contains <inspector-descriptor> elements. You can specify base inspector, which will be loaded before the inspector set itself, set using **base** attribute of **<inspector-set>** element. **base** attribute which shall contain URL of base inspector set. Using **base** attribute you can define hierarchy of inspector sets. E.g. base inspector set contains common inspectors and inspector set for web applications contains servlet-specific inspectors.
Inspector descriptor element supports the following subelements:

- **fix-sample –** Code snippet demonstrating how to fix violation reported by the inspector.
- **message** – Message which would appear in violation. It can contain formatting placeholders as described in **java.text.MessageFormat** class documentation. E.g. "Cyclomatic complexity {0} exceeds {1}". **message** element supports optional **key** attribute. Using keyed messages and org.hammurapi.InspectorContext.reportViolationEx() methods you can provide different messages from one inspector. If message element is missing **description** is used as **message.**
- **name –** Inspector name. Mandatory.
- **order –** Integer, use this if you want to impose order of invocation of visit()/leave() methods. E.g. If you want InspectorA.visit(VariableDefinition) be invoked before InspectorB.visit(VariableDefinition) and InspectorA.leave(VariableDefinition) be

invoked after InspectorB.leave(VariableDefinition) set order of InspectorA less than order of InspectorB.leave

- **rationale –** Description why inspector is needed.
- **category –** Inspector category. E.g. “Coding standards”
- **resources –** List of books, articles, URL's for additional reading.
- **severity** – Positive number. 1 corresponds to highest severity. Severity >=5 is considered 'advice' and is not counted in Sigma and DPMO calculations.
- **Violation-sample** – Code sample demonstrating code which violates standard being checked by the inspector.
- **enabled** - “yes” means that inspector is “on”, any other value means that it is “off”
- **waivable -** “yes” means that findings of this inspector can be waived either manually (through waivers file) or automatically (see chapter 10 “Auto waivers“ on page 16). Waiver stubs are generated only for waivable inspectors.
- **description** – Description of the inspector. It appears in inspectors summary. It differs from **message** because **message** is used to describe particular instance of violation and can be formatted.
- **inspector –** defines inspector class and its configuration. See chapter 21 “Instantiation and configuration of objects with DomConfigFactory“ on page 28 for more details.
- **parameter –** Inspector parameter. Inspector element supports nested parameters as well. Parameters defined at <inspector-descriptor> level can be useful in situations where you define <inspector> element in base inspector set and <parameter> element(s) in derived.
- **waive-case –** Description in which situation(s) inspector finding can be waived. Inspector author or architect shall provide waive-cases for reviewers and developers.
- **waives –** Defines which inspector this inspector can autowaive. Attribute **key** shall contain inspector 'logical' name to be used in InspectorContex.waive() method. The element itself shall contain inspector name.
- **filter –** Specifies which inspectors to filter. Attributes:
  - **name –** Inspector name to filter. * matches all inspectors**,**
  - **category** – Inspector category to filter. You shall specify on of the two.
- **filter-exclude –** Specifies which inspectors exclude from filtering. Attributes:
  - **name –** Inspector name to filter. * matches all inspectors**,**
  - **category** – Inspector category to filter. You shall specify on of the two.

Sample inspector descriptor

```
<inspector-descriptor>
  <name>ER-000</name>
  <enabled>yes</enabled>
  <severity>3</severity>
  <inspector type="org.hammurapi.inspectors.JavaLangImportRule"/>
  <description>No need to import classes from java.lang</description>
   <violation-sample><![CDATA[import java.lang.Long;]]>
</violation-sample>
</inspector-descriptor>
```

The enabled-flag control the usage.

Several inspectors needs input parameter. Please do not change the parameter names, but feel free to adapt the values. You could e.g. change the criteria for the NCSS inspector:

```
  <parameter name="function-max-loc">50</parameter>
  <parameter name="class-max-loc">500</parameter>
  <parameter name="class-max-function">20</parameter>
```

You could use a different configuration file if you manipulate the HammurapiTask method *loadEmbeddedInspectors*.
The look & feel of you reports is configured by XSL Templates. They are also part of the package. Some XSLT contains CSS.
You can build a new JAR after your modifications or you reference your runtime to the directory.

13 Hammurapi Testing

The Jsel Meta Model is constructed out of a ANTLR generated parse tree. Therefore, Junit test case are hard to apply because you have to construct a AST in order to get the Jsel objects. Testing is done with sample source code in the folder test_cases. You will find here violations and fixes. Run Hammurapi against this folder and check the report.

14 Understanding InspectorContext

Inspectors communicate with Hammurapi runtime through org.hammurapi.InspectorContext. If your inspector extends org.hammurapi.BaseInspector then you can use inherited **context** variable. Inspector can use the following methods of org.hammurapi.InspectorContext :

- addMetric(SourceMarker source, String name, double value) – invoke this method to add a metric. Metric will appear in summary, package and compilation unit reports

Metrics

Name	Number	Min	Avg	Max	Total
Class complexity	266	0.00	2.37	16.00	633.00
Code length	431	0.00	3.80	43.00	1638.00
File length	254	0.00	54.94	116.00	13956.00
NCSS Class Inspector	271	0.00	17.10	80.00	4635.00
NCSS Function Inspector	417	0.00	3.80	43.00	1587.00
Operation complexity	431	0.00	1.47	13.00	637.00

- annotate(Annotation) – to add Annotation (see chapter 16)
- debug(), info(), verbose() - to output logging messages
- getStorage() - returns `com.pavelvlasov.persistence.CompositeStorage` which is composed of three storages: `JdbcStorage` with key '`jdbc`', `FileStorage` with key '`file`' and in-memory storage with key '`memory`'. Inspector can use storage to accumulate findings to, say, build an annotation, without holding significant amount of data in memory. Class `org.hammurapi.inspectors.sample.CollectStringLiterals`

demonstrates how to use `JdbcStorage`, create annotations and use inspector lifecycle methods init() and destroy()

- reportViolation() methods - to report violations
- waive() - to waive finding of another inspector (see chapter 10)
- warn() – to output warning. Warning shall be issued if inspector couldn't complete its task. Warnings appear on Summary page and invalidate code quality metrics (DPMO and Sigma).

15 Writing your own Inspectors

If you check some of the inspector implementation you may feel motivated to implement own inspector. Please follow these steps:

1. Subclass org.hammurapi.BaseInspector
2. If you want to use parameters implement the interface of com.pavelvlasov.config.Parameterizable
3. Add a inspector-descriptor in inspector.xml. It's recommended to use for your development only a single inspector-descriptor.
4. Implement the inspector based on the visitor pattern. Implement visit(<type of interest>) for every type you want to inspect. E.g. you are looking for particular method invocation. You shall implement visit(MethodCall). Please note that if visit method returns false or Boolean.FALSE then subsequent inspectors will not be processed. It is a special case. So always make visit() methods void. You can also implement leave(<type of interest>) method. leave() methods are invoked after all visit methods for the current element and its children have been invoked. It can be useful for collecting statistics. Example (just an example): You want to enforce that any operation contains not more than 50 method calls. In this case you would write the following inspector:

```
public class MethodCallInspector extends BaseInspector {
  private Stack operationStack=new Stack();

  public void visit(Operation operation)  {
    operationStack.push(new int[] {0});
  }

  public void leave(Operation operation) {
    int[] counter=(int[]) operationStack.pop();
    if (counter[0]>50) {
      context.reportViolation(operation);
    }
  }

  public void visit(MethodCall methodCall) {
    ++((int[]) operationStack.peek())[0];
  }
}
```

   Note that we have to use stack because methods can contain local and anonymous classes.
5. Provide test case sources in folder test_cases
6. Run a self-review

Consider the memory impact of your application. Do not (hard) reference any Jsel objects, because the memory model want to garbage collect unused parts (Weak

Reference) and reload them when needed. If you need to keep references to repository members you can use one of the following approaches:

- Wrap Jsel objects or collections of objects using com.pavelvlasov.wrap.WrapperHandler.wrap(Object) method. It will create a proxy for the object you need and there will be no strong reference to the original object in your inspector.
- Set wrap attribute of Hammurapi task to true, or add –W to the command line. Hammurapi will wrap the whole model so you don't need to invoke WrapperHandler.wrap() in your code.
- Use LanguageElement.getSignature() and keep language element signature instead of a strong reference to the element. When you need that element call Repository.findBySignature().

16 Writing annotations

Annotation is a means to customize Hammurapi report by adding arbitrary information. There are two type of annotations – Linked annotations (com.pavelvlasov.results.LinkedAnnotation) and Inline annotations (com.pavelvlasov.results.InlineAnnotations). As names suggest the first is rendered as a link and the second is inlined into report page.
The sample code below shows how to create a LinkedAnnotation. You can find a sample of creating an InlineAnnotation in the source code of SimpleAnnotationSample as well.

```
public class SimpleAnnotationSample extends BaseInspector  {

  public void visit(final CompilationUnit compilationUnit) {
    context.annotate(new LinkedAnnotation() {
      private String path;
      private String cuPath=compilationUnit.getRelativeName();

      public String getName() {
        return getContext().getDescriptor().getName();
      }

      public String getPath() {
        return path;
      }

      public void render(final AnnotationContext context) throws HammurapiException {
        final AnnotationContext.FileEntry linkEntry=context.getNextFile(".txt");

        try {
          Writer out=new FileWriter(linkEntry.getFile());
          try {
            out.write("Hello, world!!!");
          } finally {
            out.close();
          }
        } catch (IOException e) {
          throw new HammurapiException("Cannot save
"+linkEntry.getFile().getAbsolutePath(), e);
        }

        AnnotationContext.FileEntry
fileEntry=context.getNextFile(context.getExtension());
        path=fileEntry.getPath(); // This file is the entry point to the annotation.
        try {
          Writer out=new FileWriter(fileEntry.getFile());
          try {
```

```
              out.write("<HTML><BODY><H1>My simple annotation</H1>"+
                   "PI: "+pi+
                   "<P><a href=\""+linkEntry.getPath()+"\">Greeting</a>" +
                   "</BODY></HTML>");
            } finally {
              out.close();
            }
          } catch (IOException e) {
            throw new HammurapiException("Cannot save
"+linkEntry.getFile().getAbsolutePath(), e);
          }
        }

        public Properties getProperties() {
          return null;
        }
      });
    }

    private Double pi;

    public void setParameter(String name, Object parameter) throws ConfigurationException {
      if ("pi".equals(name)) {
        pi=(Double) parameter;
      } else {
        throw new ConfigurationException("Parameter "+name+" is not supported");
      }
    }

    public String getConfigInfo() {
      return "PI="+pi;
    }

}
```

17 How it works, tips for plugin developers

This chapter describes how HammurapiTask.execute() method works and provides guidelines for developers who want to embed Hammurapi or write a plugin for IDE such as Eclipse or NetBeans. Figure 8 shows main players in the Hammurapi game and their top-level interaction.

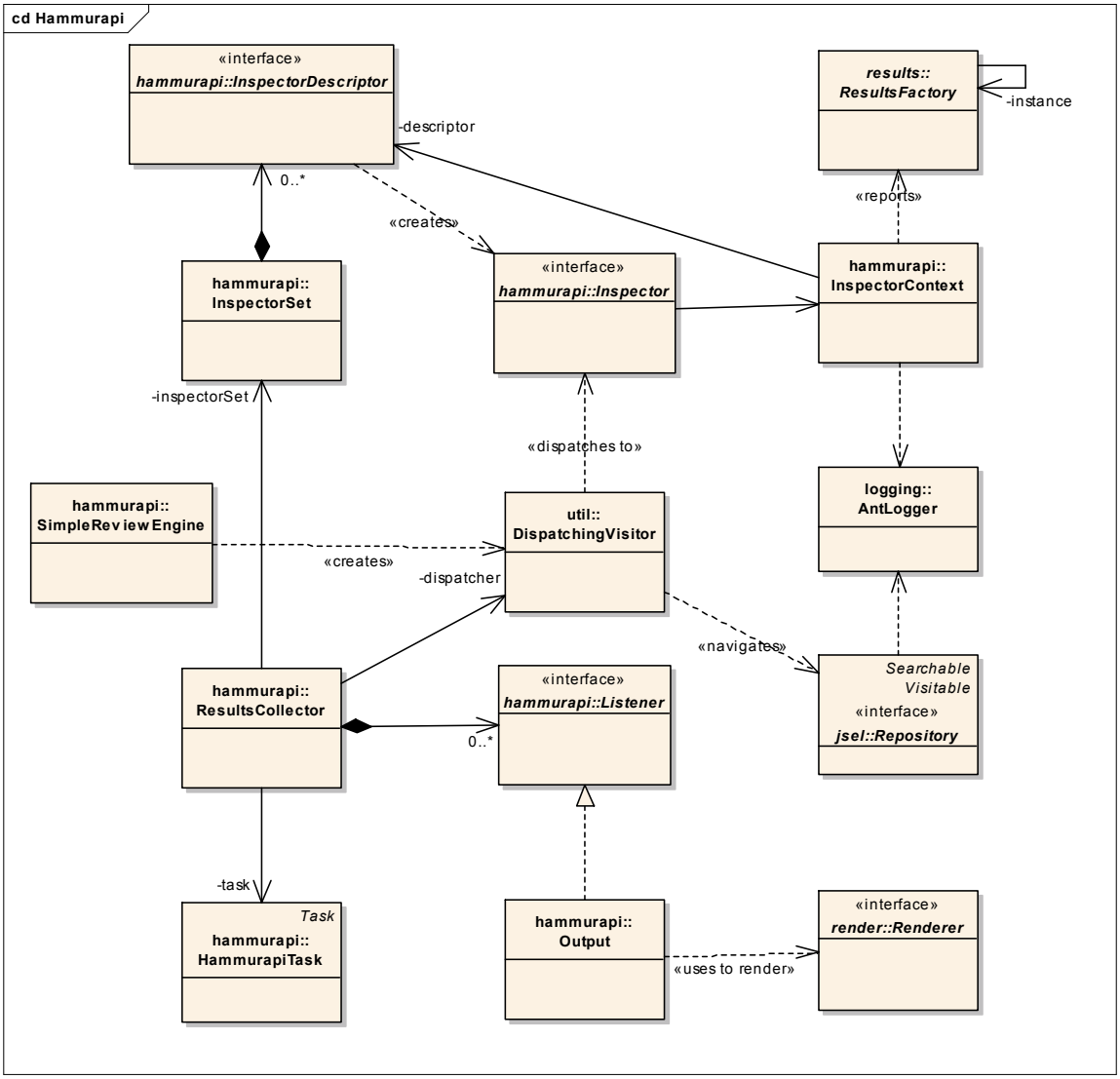

Figure 8

17.1 HammurapiTask.execute()

This section describes how execute() method works with remarks on what needs to be changed for embedding/plugging-in

1. An instance of com.pavelvlasov.logging.AntLogger created. AntLogger implements com.pavelvlasov.Logger interface so in your code you either implement this interface or use one of standard implementations from com.pavelvlasov.logging package.
2. An instance of InspectorSet is created. InspectorSet is a collection of InspectorDescriptor's
3. If embeddedInspectors is set to true then embedded inspectors from /org/hammurapi/inspectors.xml resource are loaded to the inspector set.
4. Inspector from inspector sources and inspector entries are loaded to the inspector set. Currently there is only one useful implementation of InspectorSource – DomInspectorSource. InspectorSourceEntry is a proxy to custom

InspectorSources. If you want to load inspectors from, say, database, provide an implementation of org.hammurapi.InspectorSource which reads inspectors from the database.

5. WaiverSet is instantiated.
6. Waivers are loaded to WaiverSet from WaiverSources. There is DomWaiverSource class to read waivers from XML. If you want to load waivers from other source you need to provide your own implementation of WaiverSource.
7. Listeners are loaded. Listeners are the means to receive review notifications such and must implement org.hammurapi.Listener interface.
8. Outputs are loaded. Output is just an implementation of listener which renders results to HTML/XML. You can provide your own listener which will, say, render results to Eclipse "Problems" window.
9. RepositioryConfig is created and configured. You need to set classloader, source directories, …
10. Repository is created from RepositoryConfig. Currently Jsel repository doesn't support notion of dependency, so it is as stupid as Sun javac comparing to Jikes. Repository need a DataSource where it saves parsed data. Currently HypersonicTmpDataSource is used but in the future an option will be added to specify which DataSource to use. In case of a plugin when all files are available in compiled form as well as in source form it seems to be wise to use HypersonicInMemoryDataSource and not to keep all project information in the database but only for the currently analyzed file. The rest of info shall be obtained from the classpath which shall include all compiled project files. So the idea is that you scan one file, gather results, clean the database and go to the next file. If you take this approach then do not add all project files to the config at step 9, but only the file you are going to review.
11. ResultsFactory configured and instantiated. HammurapiTask.execute() uses org.hammurap.results.persistent.jdbc.ResultsFactory to be able to store huge amount of results. For plugin you probably don't need it and you can use org.hammurapi.results.simple.SimpleResultsFactory. In this case results will remain in memory.
12. ResultsCollector is instantiated. It is a special case of inspector because it implements OrderedTarget interface. ResultsCollector's visit() methods are invoked before any inspector's visit methods and ResultCollector's leave() methods are invoked after any inspector's leave() method. This allows ResultCollector to install thread results in ResultsFactory in visit() methods before any inspector reported anything and iterate through listeners in leave() methods after all findings have been reported.
13. SimpleReviewEngine created. It does all the work inside the constructor.
14. Waiver stubs are written to the disk.
15. If any of inspector threw an exception during execution then BuildException is thrown. In plugin scenario exceptions shall be output to problem view or to the log file.

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

17.2 Plugin developer recommendations

Configuration information such as inspector set, waiver set, logger, listeners, results collector and ResultsFactory can be instantiated on load time and remain in memory for all the time while IDE runs.
Hammurapi plugin will be event driven acting on file change notifications. Jsel is not multithreaded and embedded/in-memory Hypersonic supports only one instance. Thus you can't run reviews in parallel. And in most cases there is no reason to do so as majority of developers don't use multiprocessor machines. It is my personal opinion, maybe I'm the only one guy with single-processor machine. Anyway there is no need in multithreaded reviews from IDE unless you have more than 8 processors.
So you'll need to have a background review thread and a review queue. IDE shall notify the plugin when a particular source file was changed, saved and successfully compiled after saving. Plugin will put the file name into the queue and notify the review thread. Review thread shall pick the file from the queue, clean in-memory DB and perform review by instantiating a repository with only one file in it and then a SimpleReviewEngine.

18 Customizing reports style

Reports look and feel can be customized by parameterizing embedded XSLT stylesheets or by replacing them with custom stylesheets or by the both means at the same time.
`<output>` element supports the following stylesheets:

- compilation-unit
- summary
- left-panel
- package
- inspector-set
- inspector-descriptor
- inspector-summary
- metric-details

There are two ways to replace embedded stylesheets with custom ones:

- Put custom stylesheets to classpath before `pvcommons.jar` and `hammurapi.jar`. This is the only way available to customize stylesheets in command-line mode.
- Specify stylesheet name in name attribute of nested `<stylesheet>` element of `<output>` element of Hammurapi Ant task.

Stylesheets can be parameterized by providing nested `<parameter>` element(s) to `<stylesheet>` element.

```
<stylesheet name="summary">
  <parameter name="newMarker">&lt;img
src='http://www.hammurapi.org/new.gif' alt='NEW!'/&gt;</parameter>
</stylesheet>
```

The `build.xml` fragment above parameterizes `summary` stylesheet to output

icon next to modified files and packages on summary page. `left-panel` and `package` stylesheets also support `newMarker` parameter. See Ant task documentation for more details.

Stylesheet parameterization is not available in command-line mode.

19 Incremental reviews

To run incremental reviews you need to:

- Use permanent database. Use `database` attribute or nested `server` element of Hammurapi Ant task to specify database.
- Use the same `title` attribute for reviews.
- Set `newMarker` parameter for `summary`, `left-panel` and `package` stylesheets to highlight modified files (optional)

To review all files including unchanged (e.g. if you added new inspectors to inspector set) set `force` attribute to true.

19.1 Comply-on-touch

Incremental reviews allow you to implement comply-on-touch policy, which can be very useful for legacy codebases if your organizational policy prohibits deployment of code with high severity violations to production environments and legacy code has tons of such violations because it was written before your organization adopted automated code review practice.

This policy means that you perform initial review of the code either of the following:

- empty inspector set,
- inspector set with a small number of inspectors,
- a single inspector which doesn't perform any review but outputs a low severity violation saying “Legacy code, no review performed” for every file in the repository.
- inspector set with all inspectors waived.

After that you use your standard inspector set for reviews. As only modified files will be reviewed, only those files must comply with coding standards on production move. Thus this policy ensures that quality of your code does improve and saves you from issuing tons of waivers.

20 Sample code

Class `org.hammurapi.inspectors.sample.CollectStringLiterals` demonstrates how to use `JdbcStorage`, create annotations and use inspector lifecycle methods init() and destroy()

21 Instantiation and configuration of objects with DomConfigFactory

This chapter repeats com.pavelvlasov.config.DomConfigFactory JavaDoc documentation. Check http://www.pavelvlasov.com/products/Common/doc/api/com/pavelvlasov/config/DomConfigFactory.html for up-to-date information.

com.pavelvlasov.config.DomConfigFactory reates and configures objects from DOM `Element` (XML file). DOM Element can be read from InputStream, File or URL. Instantiation and configuration happens as follows:

Instantiation

- If there is no `'type'` attribute then type defaults to `String` and text of the element will be returned. E.g. <name>Pavel</name> will yield string 'Pavel'. `'type'` attribute name can be changed through DomConfigInfo.setCodeExpression(String) method. Create DomConfigInfo, change code expression and then use DomConfigFactory(DomConfigInfo) to instantiate DomConfigFactory.
- Otherwise class specified in `'type'` attribute will be loaded and verified by classAcceptor (if any)
- If there is no nested `'constructor''` element and element text is blank then default constructor will be used
- If there is no nested `'constructor''` element and element text is not blank then constructor which takes a single argument of type java.lang.String will be used
- If there is nested `'constructor'` element then `'arg'` elements of `'constructor'` element are iterated to create a list of arguments. Arguments are constructed in the same way as described here. `'arg'` element also supports `'context-ref'` attribute. If this attribute is set argument value will be taken from context entry set by setContextEntry(String, Object) method

Configuration

- If element has attribute `'url'` and instantiated object (instance) is instance of URLConfigurable then URLConfigurable.configure(URL, Map) is invoked to configure instance
- If element has attribute `'file'` and instance is instance of FileConfigurable then FileConfigurable.configure(File, Map) is invoked to configure instance
- If instance is instance of InputStreamConfigurable then
  - If element has attribute `'url'` then that url is opened as InsputStream
  - If element has attribute `'file'` then that file is opened as InputStream
  - If element has attribute `'resource'` then that resource is opened as InputStream. Instance's class is used to obtain resource which allows to use relative resource names.

  then that InputStream is passed to InputStreamConfigurable.configure(InputStream, Map) to configure instance. If none of aforementioned attributes is present then ConfigurationException is thrown.
- If instance is instance of DomConfigurable then
  - If element has attribute `'url'` then that url is opened as InsputStream and parsed to DOM tree
  - If element has attribute `'file'` then that file is opened as InputStream and parsed to DOM tree
  - If element has attribute `'resource'` then that resource is opened as InputStream and parsed to DOM tree. Instance's class is used to obtain resource which allows to use relative resource names.

comments & bug reports appreciated: support@hammurapi.org; Johannes.Bellert@gmx.de

then that parsed document is passed to DomConfigurable.configure(Node, Map). If none of the aforementioned attributes is present then element itself is passed to DomConfigurable.configure(Node, Map)

- If instance is instance of Parameterizable then
  - If there are subelements `'parameter'` with attribute `'name'` then value of `'name'` is used as parameter name
  - Otherwise names of nested elements used as parameter names

  Parameter values are evaluated in the same way as `'arg'` elements for constructors. Parameterizable.setParameter(String, Object) is invoked for each of parameter elements. Parameterizable.setParameter(String, Object) is also invoked for context entries with names which did not match with parameter names. E.g. if there are two context entries 'age' and 'name' and parameter 'name' then setParameter("name", *value of parameter 'name'*) will be invoked and after that setParameter("age", *value of context entry 'age'*) will be invoked.
- If instance is instance of StringConfigurable then element text is passed to StringConfigurable.configure(String, Map) method
- If instance is instance of `Map` then `'entry'` subelements are iterated; `'key'` (Configurable through DomConfigInfo) and `'value'` (Configurable through DomConfigInfo) subelements are evaluated in the same way as `'arg'` constructors subelements and put to instance by `Map.put(java.lang.Object, java.lang.Object)`
- If instance is instance of `Collection` then `'element'` subelements are iterated, elements are istantiated in the same way as constructor arguments and then placed into instance by invoking `Collection.add(java.lang.Object)` method.
- If none of above conditions are met then reflection is used to set instance fields/properties in the same way as parameters for Parameterizable are set
- If object acceptor is not null then its ObjectAcceptor.accept(Object) is invoked to validate that object has been constructed and configured correctly
- If instance is instance of Validatable then Validatable.validate() is invoked for the instance to validate itself.

Examples

1. `<name>Pavel</name>` will yield java.lang.String with value 'Pavel'
2. `<age type="java.lang.Integer">33</age>` will yield java.lang.Integer with value '33'
3. `<config type="org.myself.myproject.MyConfig" url="http://myproject.myself.org/MyConfig.xml"/>` will load configuration from URL and configure MyConfig object
4.
```
<config type="org.myself.myproject.MyParameterizableConfig">
      <parameter name="pi"
type="java.lang.Double">3.14159</parameter>
 </config>
```
   will create MyParameterizableConfig object and then invoke its setParameter() method if MyParameterizableConfig implements Parameterizable or invoke setPi() method if there is such method. In lenient mode nothing will happen if there is no setPi() method. Otherwise exception will be thrown.
5. `<config type="org.myself.myproject.MyParameterizableConfig">`

```
    <pi type="java.lang.Double">3.14159</pi>
</config>
```

same as above.