kieker Documentation

Release 1.15

Various

Jan 10, 2023

Contents

Kieker is a Java-based application performance monitoring and dynamic software analysis framework. Monitoring adapters for other platforms, such as C, C++, Visual Basic~6~(VB6), .NET, and COBOL, exist as well (Contact us directly if you are interested in Kieker support for other platforms).

A general introduction can be found in *Introduction*.

CHAPTER 1

Table of Contents

1.1 Introduction

The figure below the framework's composition based on the two main components *KiekerMonitoringPart* and *KiekerAnalysisPart*.

- The *KiekerMonitoringPart* component is responsible for program instrumentation, data collection, and logging. Its core is the *MonitoringController*.
- The component *KiekerAnalysisPart* is responsible for reading, analyzing, and visualizing the monitoring data. Its core is the *AnalysisController* which manages the life-cycle of the pipe-and-filter architecture of analysis plugins, including monitoring readers and analysis filters.

Please note that older programs might use a *AnalysisController* setup while new analyses and tools reply on architecture-java-analysis-and-tools-api.

- In case you want to learn how to apply Kieker to a Java application, you find an tutorial under *Getting Started*.
- For more advanced uses you may consult *Tutorials*
- All tools are documented under *Kieker Tools*
- More documentation and API and other programming languages can be found below

1.1.1 Framework Components and Extension Points

Fig. 1: Kieker framework components and extension points for custom components

The Figure above depicts the possible extension points for custom components as well as the components which are already included in the **Kieker** distribution and detailed below.

- **Monitoring writers and corresponding readers** for file systems and SQL databases, for in-memory record streams (named pipes), as well writers and readers employing Java Management Extensions (JMX) and Java

Messaging Service (JMS) technology. A special reader allows to replay existing persistent monitoring logs, for example to emulate incoming monitoring data—also in real-time.

- **Time sources** utilizing Java's `System.nanoTime()` (default) or `System.current\-TimeMillis()` methods.
- **Monitoring record types** allowing to store monitoring data about operation executions (including timing, control-flow, and session information), CPU and resource utilization, memory/swap usage, as well as a record type which can be used to store the current time.
- **Monitoring probes**: A special feature of **Kieker** is the ability to monitor (distributed) traces of method executions and corresponding timing information. For monitoring this data, Kieker includes monitoring probes employing AspectJ, Java EE, Servlet, Spring, and Apache CXF technology. Additionally, Kieker includes probes for (periodic) system-level resource monitoring employing OSHi.
- **Analysis/Visualization plugins** can be assembled to pipe-and-filter architectures based on input and output ports. The **KiekerTraceAnalysis** tool is itself implemented based on Kieker Analysis filters allowing to reconstruct and visualize architectural models of the monitored systems, e.g., as dependency graphs, sequence diagrams, and call trees.

1.2 Getting Started

In this section we introduce how to work with Kieker. Starting with methods to obtain Kieker, configure and apply Kieker, and writing your own probes and event types. We use the small sample application Bookstore to illustrate how to work with Kieker. However, there exists a wide variety how to apply Kieker to applications and services. We will only cover manual and basic AspectJ instrumentation to discuss basic concept. There is more documentation available (how-to-apply-insturmentation) which illustrates different techniques to instrument depending on the technology.

- gt-download-and-extract-tutorial
- gt-the-bookstore-example-application
- gt-manual-monitoring-with-kieker
- gt-aspectj-instrumentation-example
- gt-using-kieker-trace-analysis

1.3 Tutorials

Collection of reoccurring tasks when using Kieker.

1.3.1 How to Apply Insturmentation

1.3.2 How to contribute Documentation

Todo: This must be replaced to fit the current docuemtnation with restructured text.

1.3.3 Java Servlet Container Example

Using the sample Java web application MyBatis *JPetStore <http://www.mybatis.org/spring/sample.html>* this example demonstrates how to employ Kieker for monitoring a Java application running in a Java Servlet container – in this case *Jetty <http://www.eclipse.org/jetty/>*. Monitoring probes based on the Java Servlet API, Spring and AspectJ are used to monitor execution, trace, and session data (see also **instrumenting-software-aspectj_**).

Prerequisites

- Download and extract the ***Kieker** binary distribution <http://kieker-monitoring.net/download/>*
- The directory `kieker-1.14/examples/JavaEEServletContainerExample` contains the prepared Jetty server with the MyBatis JPetStore application and the **Kieker**-based demo analysis application known from the *Kieker Homepage <http://demo.kieker-monitoring.net/>*.
- Switch to this directory or copy it to a suitable location.

Instrumenting Servlets

The subdirectory `jetty` includes the Jetty server with the JPetStore application already deployed to the server's `webapps/` directory. The example is prepared to use two alternative types of **Kieker** probes: either the **Kieker** Spring interceptor (default) or the **Kieker** AspectJ aspects. Both alternatives additionally use **Kieker**'s Servlet filter.

Required Libraries and Kieker Monitoring Configuration

Both settings require the files `aspectjweaver-1.8.2.jar` and `kieker-1.14`, which are already included in the webapps's `WEB-INF/lib/` directory. Also, a **Kieker** configuration file is already included in the Jetty's root directory, where it is considered for configuration by Kieker Monitoring in both modes.

Servlet Filter (Default)

The file `web.xml` is located in the webapps's `WEB-INF/` directory. **Kieker**'s Servlet filters are already enabled:

```
<filter>
  <filter-name>sessionAndTraceRegistrationFilter</filter-name>
  <filter-class>kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter</
↪filter-class>
  <init-param>
    <param-name>logFilterExecution</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>sessionAndTraceRegistrationFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

This filter can be used with both the Spring-based and the AspectJ-based instrumentation mode.

Spring-based Instrumentation (Default)

Kieker's Spring interceptor are already enabled in the file `applicationContext.xml`, located in the webapps's `WEB-INF/` directory:

```
<!-- Kieker's instrumentation probes based on the Spring AOP interception framework --
↪>
<bean id="opEMII"
      class="kieker.monitoring.probe.spring.executions.
↪OperationExecutionMethodInvocationInterceptor" />
<aop:config>
   <aop:advisor advice-ref="opEMII"
                pointcut="execution(public * org.mybatis.jpetstore..*.*(..))"/>
</aop:config>
```

Note: When using, for example, the `@Autowired` feature in your Spring beans, it can be necessary to force the usage of CGLIB proxy objects with `<aop:aspectj-autoproxy proxy-target-class="true"/>`.

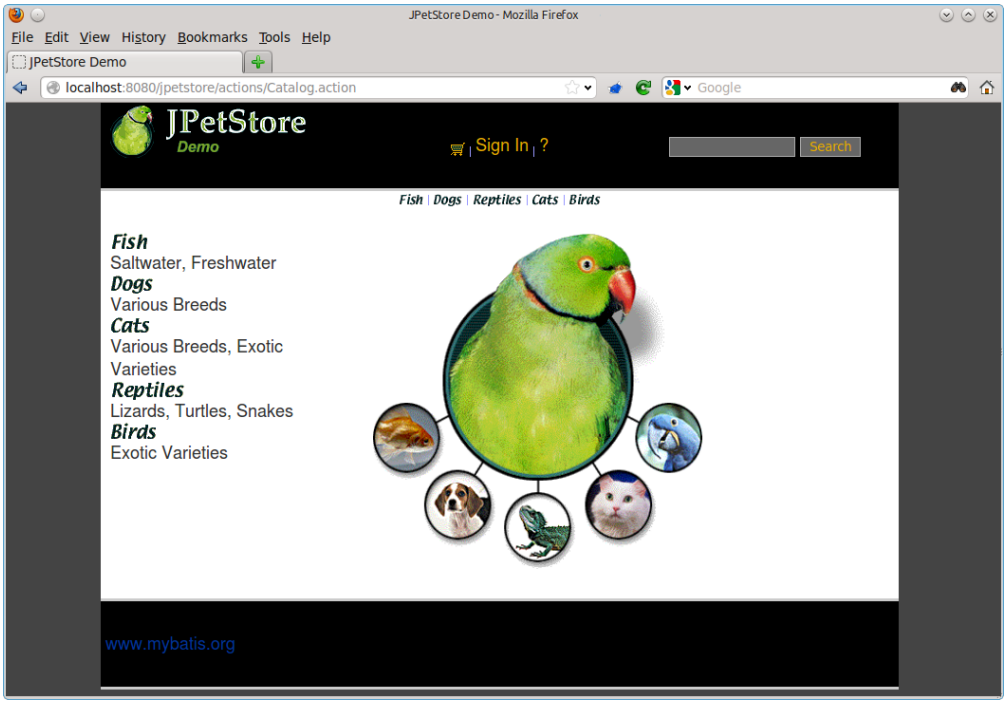
AspectJ-based Instrumentation

In order to use AspectJ-based instrumentation, the following changes need to be performed. The file `start.ini`, located in Jetty's root directory, allows to pass various JVM arguments, JVM system properties, and other options to the server on startup. When using AspectJ for instrumentation, the respective JVM argument needs to be activated in this file.

The AspectJ configuration file `aop.xml` is already located in the webapps's `WEB-INF/classes/META-INF/` directory and configured to instrument the JPetStore classes with **Kieker**'s **OperationExecutionAspectFull** aspect.

When using the AspectJ-based instrumentation, make sure to disable the Spring interceptor in the file `applicationContext.xml`, mentioned above.

1. Start the Jetty server using the `start.jar` file (e.g., via `java -jar start.jar`). You should make sure that the server started properly by taking a look at the console output that appears during server startup.
2. Now, you can access the JPetStore application by opening the URL http://localhost:8080/jpetstore/. **Kieker** initialization messages should appear in the console output.



3. Browse through the application to generate some monitoring data.
4. In this example, **Kieker** is configured to write the monitoring data to JMX in order to communicate with the **Kieker**-based demo analysis application, which is accessible via <localhost:8080/livedemo/<.
5. In order to write the monitoring data to the file system, the JMX writer needs to be disabled in the file `kieker.monitoring.properties`, which is located in the directory `webapps/jpetstore/WEB-INF/classes/META-INF/`. After a restart of the Jetty server, the Kieker startup output includes the information where the monitoring data is written to (should be a `kieker-<DATE-TIME>/` directory) located in the default temporary directory. This data can be analyzed and visualized using **kieker-tools-trace-analysis-tool_**.

1.3.4 How to apply Kieker in Java EE Environments

Depending on the configuration of the JavaEE application to be monitored, different instrumentation technologies (e.g., Spring AOP and SOAP/CXF interceptors) can and should be used. In this page, we show how to use AspectJ to monitor method calls in different JavaEE environments.

Jetty

Copy `kieker-1.15_aspectj.jar` into a directory, where it can be accessed by Jetty, e.g. `jetty/kieker/`.

Jetty is usually shipped with a configuration file start.ini in which start parameters are defined. Add the following snippet to this file:

```
--exec
-javaagent:kieker/kieker-1.15_aspectj.jar
-Dkieker.monitoring.skipDefaultAOPConfiguration=true
-Daj.weaving.verbose=true
```

To use a custom Kieker configuration at the given location, the following parameter can be added:

`-Dkieker.monitoring.configuration=kieker/kieker.monitoring.properties`

Important: Due to a bug in the parser of Jetty, a line ending with .properties is misinterpreted and leads to an exception. We recommend to rename the extension of Kieker's configuration file.

To use a custom AspectJ configuration at the given location, the following parameter can be added:

`-Dorg.aspectj.weaver.loadtime.configuration=file://c:/jetty/kieker/aop.xml`

Important: There seems to be problems with relative paths for AspectJ in JavaEE environments. We recommend to use URIs instead.

Tested with JPetStore 6 and Jetty 9.2.2.

JBoss

- Needs documentation but ' NovaTec's blog post <http://blog.novatec-gmbh.de/analysing-kieker-with-jboss-dvdstore-sample-application/>'__ may serve as a starting point

1.3.5 Tomcat

Copy `kieker-1.10_aspectj.jar` into a directory, where it can be accessed by Tomcat, e.g. `tomcat/kieker/`.

Tomcat is usually shipped with start scripts `bin/catalina.(sh|.bat)`. Add one of the following snippets to the correct file, depending on your operation system:

```
set JAVA_OPTS=%JAVA_OPTS%
        -javaagent:%CATALINA_BASE%\kieker\kieker-1.10_aspectj.jar
        -Dkieker.monitoring.skipDefaultAOPConfiguration=true
        -Daj.weaving.verbose=true
        -Dkieker.monitoring.configuration=%CATALINA_BASE%/kieker/kieker.monitoring.
↪properties
        -Dorg.aspectj.weaver.loadtime.configuration=...

set JAVA_OPTS=${JAVA_OPTS}
        -javaagent:${CATALINA_BASE}\kieker\kieker-1.10_aspectj.jar
        -Dkieker.monitoring.skipDefaultAOPConfiguration=true
        -Daj.weaving.verbose=true
        -Dkieker.monitoring.configuration=${CATALINA_BASE}/kieker/kieker.monitoring.
↪properties
        -Dorg.aspectj.weaver.loadtime.configuration=...
```

...

- Needs documentation but ticket/566 may serve as a starting point

1.3.6 Glassfish

On Glassfish 4, this can be achieved with adding properties to the domain.xml. The default domain in glassfish is normally called domain0. Let further assume that glassfish was installed in /opt/glassfish-4.0 then the domain.xml file will be located in /opt/glassfish-4.0/glassfish/domains/domain0/config. In that file search for jvm-options. You will find multiple such entries between

```
<java-config ...>
        <jvm-options>...</jvm-options>
</java-config>
```

After the last entry in that XML environment, add the following lines and adapt the paths to your situation.

```
<jvm-options>-javaagent:${com.sun.aas.installRoot}/lib/kieker-1.15_aspectj.jar</jvm-
↪options>
<jvm-options>-Dkieker.monitoring.skipDefaultAOPConfiguration=true</jvm-options>
<jvm-options>-Daj.weaving.verbose=true</jvm-options>
<jvm-options>-Dkieker.monitoring.configuration=${com.sun.aas.installRoot}/kieker/
↪kieker.monitoring.properties</jvm-options>
<jvm-options>-Dorg.aspectj.weaver.loadtime.configuration=${com.sun.aas.installRoot}/
↪kieker</jvm-options>
```

1.3.7 WebSphere

- Needs documentation

1.3.8 JBoss (Wildfly)

An alternative approach to run Kieker within a JBoss environment is described here.

Requires:

- Kieker-1.13 or above (1.12 and below cause an error in JBoss environments)
- Kieker packed as Wildfly module
- AspectJ Weaver packed as Wildfly module

Kopiere beide Module einfach in das folgende Wildfly-Verzeichnis:

`modules/system/layers/base`

Kopiere die Dateien `kieker.properties` und `aop.xml` in das (neue) Verzeichnis "kieker" auf oberster Ebene von Wildfly. Passe nun den Ausgabepfad von Kieker an und schränke das Instrumentieren auf den gewünschten Paketnamen ein.

[in der Datei standalone.xml]

- Unter dem folgenden, vorhandenen Subsystem müssen die beiden, neuen Module als globale Module registriert werden:

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
<global-modules>
        <module name="kieker"/>
        <module name="org.aspectj"/>
</global-modules>
```

[in der Datei standalone.conf]

- Dort, wo die Systempakete deklariert werden, müssen die folgenden ergänzt werden: org.jboss.logmanager, com.manageengine, org.aspectj, kieker. Du musst im Folgenden nur darauf achten, dass ich Windows-Syntax für die Skriptbefehle genutzt habe.

```
set "JAVA_OPTS=%JAVA_OPTS% -Djboss.modules.system.pkgs=org.jboss.byteman,org.jboss.
↪logmanager,com.manageengine,org.aspectj,kieker"

set "WILDFLY=I:\Software-Engineering\wildfly-10.1.0.Final"
```

- Weiterhin muss der Aspectjweaver als Javaagent eingetragen werden und für Wildfly entsprechende notwendige Ergänzungen vorgenommen werden, die das Verwenden von AspectJ überhaupt erst ermöglichen:

```
set "JAVA_OPTS=%JAVA_OPTS% -javaagent:%WILDFLY%/modules/system/layers/base/org/
↪aspectj/main/aspectjweaver.jar"

set "JAVA_OPTS=%JAVA_OPTS% -Djava.util.logging.manager=org.jboss.logmanager.LogManager
↪"

set "JAVA_OPTS=%JAVA_OPTS% -Xbootclasspath/p:%WILDFLY%/modules/system/layers/base/org/
↪jboss/logmanager/main/jboss-logmanager-2.0.4.Final.jar;%WILDFLY
↪%\modules\system\layers\base\kieker\main\kieker-1.15.jar;%WILDFLY
↪%\modules\system\layers\base\org\aspectj\main\aspectjweaver.jar"
```

- Anschließend werden Einstellungen für das Monitoring durch Kieker vorgenommen:

```
set "JAVA_OPTS=%JAVA_OPTS% -Dkieker.monitoring.configuration=%WILDFLY%/kieker/kieker.
↪monitoring.1.13.properties"

set "JAVA_OPTS=%JAVA_OPTS% -Dkieker.monitoring.skipDefaultAOPConfiguration=true"

set "JAVA_OPTS=%JAVA_OPTS% -Daj.weaving.verbose=true"

set "JAVA_OPTS=%JAVA_OPTS% -Dorg.aspectj.weaver.loadtime.configuration=file:%WILDFLY%/
↪kieker/aop.xml"
```

Wenn du nun Wildfly startest, sollten keine Fehler erscheinen. Da das Instrumentieren erst beim Laden der entsprechenden Klassen erfolgt, siehst du an dieser Stelle nur Konsolenausgaben von AspectJ. Erst wenn du das Szenario ausführst, wird Kieker gestartet und der Log-Ordner angelegt und mit Daten gefüllt.

1.3.9 How to pass the monitoring configuration to Kieker

The Kieker Monitoring Controller checks several locations for the kieker configuration. Initially, Kieker tries to read `META-INF/kieker.monitoring.default.properties` file. If it cannot read this file it uses the built in defaults for the configuration. Subsequently, Kieker checks whether the `kieker.monitoring.configuration` JVM parameter is set and tries to load the configuration from there.

To provide an alternative location for a Kieker configuration in context of command line applications, please add `-Dkieker.monitoring.configuration=FULL_PATH_TO_LOCATION` to the java set of parameters, e.g., `java -Dkieker.monitoring.configuration=/myconfiguration -jar MyApplication.jar`

For war file, add your configuration to the `META-INF` folder or pass the property to the server, e.g., tomcat.

1.3.10 How to collect Traces from Servlets

While we use Jetty in this tutorial, other servlet containers can also be used. Information on them can be found in *How to apply Kieker in Java EE Environments*

Prerequisites

- Use Java 8 (it may work with newer versions, see below)
- Get the iObserve variant of the JPetStore from *JPetStore-6 <https://github.com/research-iobserve/jpetstore-6>* Use `git clone https://github.com/research-iobserve/jpetstore-6.git` to obtain the JPetStore

- Download *Jetty <https://www.eclipse.org/jetty/download.html>*. Use a version which is compatible with your Java version. Kieker 1.14 with AspectJ will not work with Java 11. This will change in the 1.15 which supports Java 11.

Instrumenting Servlets

Running an Example Application

1.3.11 How to perform Trace Analysis

Todo: Fix internal references.

Kieker *trace-analysis* implements the special feature of **Kieker** allowing to monitor, analyze, and visualize (distributed) traces of method executions and corresponding timing information. For this purpose, it includes monitoring probes employing AspectJ, Java Servlet, Spring, and Apache CXF technology. Moreover, it allows to reconstruct and visualize architectural models of the monitored systems, e.g., as sequence and dependency diagrams.

In this tutorial, we will instrument a Java Servlet application with interceptors and AspectJ. For other options to generate traces in Java and other programming languages, please consult the respective pages in *How to perform Trace Analysis* and *How to apply Kieker in Java EE Environments*.

We use the `OperationExecutionRecord` from the `controlflow` package to collect trace information. There is also an alternative `flow`-based set of monitoring events which can be used alternatively. However, they are not used in this tutorial. More information on monitoring traces can be found in *tutorials-how-to-perform-trace-analysis*.

The `OperationExecutionRecord` attributes `operationName`, `tin`, and `tout` represent the full qualified name of the operation including the class name, the time before execution of the operation and the time after the execution, respectively (see JavaDoc OperationExecutionRecord). The attributes `traceId` and `sessionId` are used to store trace and session information; `eoi` and `ess` contain control-flow information needed to reconstruct traces from monitoring data. For details please refer to the technical report and JavaDoc.

Prerequisites

- A basic understanding of how Kieker performs monitoring (see *Getting Started*)
- Basic knowledge of AspectJ, i.e., that it is an aspect-oriented approach and technology
- Basic knowledge what a Servlet application is
- Docker, in case you want to use docker to run the example (optional)
- Download the Servlet Engine Jetty (tested with 9.4.30)

Getting JPetStore

Checkout the JPetStore here and switch to the single-jpetstore branch, for a vanilla JPetStore. Please note: There is also a variant pre-configured with **Kieker** probes utilizing the flow events instead of the controlflow events used in this tutorial.

```
git clone https://github.com/research-iobserve/jpetstore-6.git

cd jpetstore-6
git checkout single-jpetstore
```

Now it is time to check whether your version compiles with

```
mvn compile package
```

This produces an output similar to

```
[INFO]
[INFO] --- maven-war-plugin:3.1.0:war (default-war) @ jpetstore ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jpetstore] in [/home/user/jpetstore-6/target/jpetstore]
[INFO] Processing war project
[INFO] Copying webapp resources [/home/user/jpetstore-6/src/main/webapp]
[INFO] Webapp assembled in [97 msecs]
[INFO] Building war: /home/user/jpetstore-6/target/jpetstore.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 19.034 s
[INFO] Finished at: 2020-06-15T13:22:23+02:00
[INFO] ------------------------------------------------------------------------
```

The resulting war file is located in `target` inside the main project directory `jpetstore-6` and named `jpetstore.war`.

Instrumenting JPetStore

JPetStore is a small demonstration example of a Servlet based application. That means external HTTP requests to the application trigger a trace through the application. Therefore, we must instrument the incoming request and all subsequent method calls through the application. Thus, we must use Servlet interceptors and instrument all methods, which we can do with AspectJ.

Instrumenting Servlet Requests

The Java Servlet API includes the `javax.servlet.Filter` interface. It can be used to implement interceptors for incoming HTTP requests. Kieker provides a `SessionAndTraceRegistrationFilter` probe which implements the `javax.servlet.Filter` interface. It initializes the session and trace information for incoming requests. If desired, it additionally creates an `OperationExecutionRecord` for each invocation of the filter and passes it to the `MonitoringController`. To integrate the interceptor into the application, you must add a filter configuration to the `web.xml` file. The web.xml file is located in `jpetstore-6/src/main/webapp/WEB-INF`

```
<filter>
        <filtername>sessionAndTraceRegistrationFilter</filtername>
        <filterclass>kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter
↪</filterclass>
        <initparam>
                <paramname>logFilterExecution</paramname>
                <paramvalue>true</paramvalue>
        </initparam>
</filter>
<filtermapping>
        <filtername>sessionAndTraceRegistrationFilter</filtername>
        <urlpattern>/</urlpattern>
</filtermapping>
```

In the above snippet, the **Kieker** class `kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter` implementing the probe is registered in the Servlet application and the `filter-mapping` assigns it to all Servlet URLs.

Instrumenting Method Calls

While the Servlet filter above will collect all HTTP requests to the application, it cannot collect the traces within the application. Therefore, we have to apply probes to all methods. In this tutorial, we use AspectJ and **Kieker**'s AspectJ probes to accomplish this goal.

Kieker includes the AspectJ-based monitoring probes `OperationExecutionAspectAnnotation`, `OperationExecu-tionAspectAnnotationServlet`, `OperationExecutionAspectFull`, and `OperationExecutionAspectFullServlet` which can be woven into Java applications at compile time and load time. These probes monitor method executions and corresponding trace and timing information. The probes with the postfix `Servlet` additionally store a session identifier within the `OperationExecutionRecord`. For this tutorial, we use `OperationExecutionAspectFull` probe to collect trace information.

To configure AspectJ, we have to create an `aop.xml` file and place it `src/main/resources` within the jpetstore-6 project directory. It contains the following lines:

```
<!DOCTYPE aspectj PUBLIC "//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/aspectj_1_5_
↪0.dtd">
<aspectj>
        <weaver options="">
                <include within="org.mybatis..*"/>
        </weaver>
        <aspects>
                <aspect name="kieker.monitoring.probe.aspectj.operationExecution.
↪OperationExecutionAspectFull"/>
        </aspects>
</aspectj>
```

Line 5 specifies which classes and methods within the project shall be instrumented. The `org.mybatis..*` limits the instrumentation to classes of the project itself and ignores all imported jar files, as we are not interested to clutter the results with API internals. Line 9 selects the aspect `OperationExecutionAspectFull`. As indicated by its name, this aspect makes sure that every method within the included classes/packages will be instrumented and monitored.

Adding Dependencies

The JPetStore example uses Maven to build the application. Therefore, we have now to adapt the build configuration to use AspectJ and Kieker. Maven is configured by a `pom.xml` file located in the project root directory.

Open the `pom.xml` in an editor. Here you must add

- the dependencies for Kieker and AspectJ, and
- the AspectJ compile time weaving.

In the dependency section of the `pom.xml` add:

```
<dependency>
        <groupId>net.kieker-monitoring</groupId>
        <artifactId>kieker</artifactId>
        <version>1.14</version>
</dependency>
```

(continues on next page)

(continued from previous page)

```
<dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.8.7</version>
</dependency>
```

In the build section of the `pom.xml` add:

```
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <version>1.8</version>
        <configuration>
                <source>1.7</source>
                <target>1.7</target>
                <complianceLevel>1.7</complianceLevel>
                <aspectLibraries>
                        <aspectLibrary>
                                <groupId>net.kieker-monitoring</groupId>
                                <artifactId>kieker</artifactId>
                        </aspectLibrary>
                </aspectLibraries>
                <xmlConfigured>${basedir}/src/main/resources/aop.xml</xmlConfigured>
                <sources>
                        <source>
                                <basedir>${basedir}/src/main/java</basedir>
                                <includes>
                                        <include>**/**.java</include>
                                </includes>
                        </source>
                </sources>
        </configuration>
        <executions>
                <execution>
                        <goals>
                                <goal>compile</goal>
                        </goals>
                </execution>
        </executions>
</plugin>
```

Please note that the `src/main/resources/aop.xml` is explicitly specified in the configuration.

Configuring Kieker

The last step is to place a **Kieker** configuration file within the application to instruct the *MonitoringController* where and how to store the monitoring data. The `kieker.monitoring.properties` file should contain the following information and must be placed in `src/main/resources/META-INF/` within the project directory.

```
## The name of the Kieker instance.
kieker.monitoring.name=KIEKER

## Whether a debug mode is activated.
kieker.monitoring.debug=false
```

(continues on next page)

(continued from previous page)

```
## Enable monitoring after startup
kieker.monitoring.enabled=true

## The name of the VM running Kieker or empty (will automatically be
resolved)
kieker.monitoring.hostname=

## Automatically add a metadata record
kieker.monitoring.metadata=true

## Enables the automatic assignment
kieker.monitoring.setLoggingTimestamp=true

## Register shutdown hook
kieker.monitoring.useShutdownHook=true

## Do not use JMX
kieker.monitoring.jmx=false

## The size of the thread pool used to execute registered periodic sensor jobs.
kieker.monitoring.periodicSensorsExecutorPoolSize=0

## Disable adaptive monitoring.
kieker.monitoring.adaptiveMonitoring.enabled=false

## Timer to use
kieker.monitoring.timer=kieker.monitoring.timer.SystemNanoTimer

## Report timestamps in
## Accepted values:
## 0 - nanoseconds
## 1 - microseconds
## 2 - milliseconds
## 3 - seconds
kieker.monitoring.timer.SystemMilliTimer.unit=0

## Writer configuration
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.FileWriter

## output path
kieker.monitoring.writer.filesystem.FileWriter.customStoragePath=$LOGGING_DIR/
kieker.monitoring.writer.filesystem.FileWriter.charsetName=UTF-8

## Number of entries per file
kieker.monitoring.writer.filesystem.FileWriter.maxEntriesInFile=25000

## Limit of the log file size; -1 no limit
kieker.monitoring.writer.filesystem.FileWriter.maxLogSize=-1

## Limit number of log files; -1 no limit
kieker.monitoring.writer.filesystem.FileWriter.maxLogFiles=-1

## Map files are written as text files
kieker.monitoring.writer.filesystem.FileWriter.mapFileHandler=kieker.monitoring.
↪writer.filesystem.TextMapFileHandler

## Flush map file after each record
```

(continues on next page)

(continued from previous page)

```
kieker.monitoring.writer.filesystem.TextMapFileHandler.flush=true

## Do not compress the map file
kieker.monitoring.writer.filesystem.TextMapFileHandler.compression=kieker.monitoring.
↪writer.compression.NoneCompressionFilter

## Log file pool handler
kieker.monitoring.writer.filesystem.FileWriter.logFilePoolHandler=kieker.monitoring.
↪writer.filesystem.RotatingLogFilePoolHandler

## Text log for record data
kieker.monitoring.writer.filesystem.FileWriter.logStreamHandler=kieker.monitoring.
↪writer.filesystem.TextLogStreamHandler

## Do not compress the log file
kieker.monitoring.writer.filesystem.TextLogStreamHandler.compression=kieker.
↪monitoring.writer.compression.NoneCompressionFilter

## Flush log data after every record
kieker.monitoring.writer.filesystem.FileWriter.flush=true

## buffer size. The log buffer size must be big enough to hold the biggest record
kieker.monitoring.writer.filesystem.FileWriter.bufferSize=81920
```

Key for the writer configuration are two properties `kieker.monitoring.writer` which selects the writer and `kieker.monitoring.writer.filesystem.FileWriter.customStoragePath` which specifies where the data shall be stored. In this tutorial, we use the `kieker.monitoring.writer.filesystem.FileWriter` which can write text and binary log files and even compress the output if necessary. If no `customStoragePath` is specified, Kieker will write to `/tmp` on Unix machines or to the respective system wide directory for temporary files. In the above code snippet, we specified $LOGGING_DIR as location for log files. Please choose an appropriate path within your system.

Build and Run

To build the example got to the project root directory and type:

```
mvn clean compile package
```

This will produce a `jpetstore.war` file located in the `target` directory of the jpetstore-6 project.

To run the JPetStore:

- Download Jetty in case you have not done this already.
- Unpack Jetty next to the `jpetstore-6` project directory, e.g.,

```
drwxr-xr-x 11 user example 4096 Jun 15 14:32 jetty-distribution-9.4.30.v20200611
drwxrwxr-x 7 user example 4096 Jun 15 13:22 jpetstore-6
```

- Copy the `jpetstore.war` to the jetty `webapps` directory

```
cp jpetstore-6/target/jpetstore.war jetty-distribution-9.4.30.v20200611/webapps
```

- Switch to the Jetty directory and start the application

```
cd jetty-distribution-9.4.30.v20200611

java -jar start.jar
```

- Now you can access the JPetStore from your browser with http://localhost:8080/jpetstore
- While you are using the application logging information appears in a newly created Kieker logging directory, e.g.,
  - `kieker-20200615-130444-341575577055999-UTC--KIEKER/`
    - `kieker-20200615-130444372-UTC-001.dat`
    - `kieker.map`
- Feel free to explore the whole JPetStore. While browsing through the shop, you will notice that the log files will grow over time.

Analyzing Traces

Monitoring data including trace information can be analyzed and visualized with the **Kieker** *trace-analysis* tool which is included in the **Kieker** binary distribution as well. A the tool outputs dot and pict files, tools to view such files are required. We usually use GraphViz and GnuPlot utils.

In order to use this tool, it is necessary to install two third-party pro-grams:

1. **GraphViz** A graph visualization software which can be down-loaded from `http://www.graphviz.org`
2. **GNU PlotUtils** A set of tools for generating 2D plotgraphics which can be downloaded from `http://www.gnu.org/software/plotutils/` (for Linux) and from `http://gnuwin32.sourceforge.net/packages/plotutils.htm` (for Windows).
3. **ps2pdf** The`ps2pdf`tool is used to convert ps files to pdf files.

Under Windows it is recommended to add the `bin/` directories of both tools to the "path" environment variable. It is also possible that the GNU PlotUtils are unable to process sequence diagrams. In this case it is recommended to use the Cygwin port of PlotUtils.

Once both programs have been installed, the **Kieker** `trace-analysis` tool can be used. It can be found in the `tools` directory of the Kieker binary release. Unpack the `trace-analysis-1.14.zip` alongside the `jpetstore-6` directory. Start scripts can then be found in `trace-analysis-1.14/bin/trace-analysis` (Unix) and `trace-analysis-1.14/bin/trace-analysis.bat` (Windows). Non-parameterized calls of the scripts print all possible options on the screen.The commands shown in Listings below generate a sequence diagram as well as a call tree to an existing directory named `out/`. The monitoring data is assumed to be located in the logging directory, e.g., `kieker-20200615-130444-341575577055999-UTC--KIEKER/` alongside the `jpetstore-6` directory.

Before executing the trace-analysis, you need to create the `out/` directory alongside the `jpetstore-6` directory.

Unix version

```
trace-analysis-1.14/bin/trace-analysis -inputdirs kieker-20200615-130444-
↪341575577055999-UTC--KIEKER \
       -outputdirout/ \
       -plot-Deployment-Sequence-Diagrams-plot-Call-Trees-short-labels
```

Windows version

```
trace-analysis-1.14/bin/trace-analysis.bat -inputdirs kieker-20200615-130444-
↪341575577055999-UTC--KIEKER
       -outputdir out\
       -plot-Deployment-Sequence-Diagrams-plot-Call-Trees-short-labels
```

The resulting contents of the `out/` directory should be similar to the following tree:

- `out/`
  - `deploymentSequenceDiagram-6120391893596504065.pic`
  - `callTree-6120391893596504065.dot`
  - `system-entities.html`

The `.pic` and `.dot` files can be converted into other formats, such as `.pdf`, by using the GraphViz and Plot Utils tools `dot` and `pic2plot`. Type the following to generate PDF file from the graphics.

```
dot callTree6120391893596504065.dot -T pdf -o callTree.pdf
pic2plot deploymentSequenceDiagram6120391893596504065.pic-T pdf > sequenceDiagram.pdf
```

The scripts `dotPic-fileConverter.sh` and `dotPic-fileConverter.bat` convert all `.pic` and `.dot` in a specified directory. The scripts can be found in the bin directory of the **Kieker** binary distribution.

Example Outputs of the Trace Analysis

1.3.12 How to configure Kieker within Java-Applications and -Services

There are three scenarios where Kieker configuration parameters can be used Java applications and must be configured on command line or startup scripts.

1. Normal Java application without Kieker parts which should be instrumented and observed
2. Java applications which has already Kieker integrated or woven in with AspectJ
3. Java application which uses Kieker directly and accepts an Kieker property file

Normal Java Application

This works largely like the second option. However, you have to add the Kieker agent to the java invocation.

Application with integrated Kieker at Compile or Bundling Time

In case you have an application which need Kieker configuration parameters set, but which does not provide a command line option for such configuration file can add `-Dkieker.monitoring.configuration=$CONFIGUATION_FILE` to the java invocation statement. In many gradle-based builds this can be achieved by using the `*_OPTS` environment variable. The * represents the name of the tool.

Kieker-based Application

Add your configuration parameters to the application's configuration file

1.3.13 How to Write Tests for Your own Kieker Probes

Writing your own probes with Kieker is quite simple. However, testing them requires additional insight into Kieker which require reading a lot of source code. As this is an unpleasant task, I collected some basic ideas in this how-to.

Let say you have a written a probe `ExampleProbe`:

```
import kieker.common.record.IMonitoringRecord;
import kieker.common.record.flow.trace.TraceMetadata;
import kieker.common.record.flow.trace.operation.BeforeOperationEvent;
import kieker.monitoring.core.controller.IMonitoringController;
import kieker.monitoring.core.controller.MonitoringController;
import kieker.monitoring.core.registry.TraceRegistry;

public class ExampleProbe {
        private final IMonitoringController ctrl = MonitoringController.getInstance();
        private final TraceRegistry registry = TraceRegistry.INSTANCE;

        public ExampleProbe() {
        }

        public void takeMeasurement(final String operationSignature,
                final String classSignature) {

                /** collect event data. */
                final TraceMetadata trace = this.registry.getTrace();
                final long timestamp = this.ctrl.getTimeSource().getTime();
                final long traceId = trace.getTraceId();
                final int orderIndex = trace.getNextOrderId();

                /** create event. */
                final IMonitoringRecord event = new BeforeOperationEvent(timestamp,
                        traceId, orderIndex, operationSignature,
                        classSignature);

                /** log event. */
                this.ctrl.newMonitoringRecord(event);
        }
}
```

When you use this in an application, the `IMonitoringController` will refer to a singleton within the application, which is great within an application. You can pass a configuration via a file at a default location or by specifying an environment variable. Unfortunately, this makes is more complicated for testers to pass their configuration to the controller. In case you instantiate a controller in the test class, it will create a separate controller for the test class. Fortunately, there is a way around this limitation. As the `MonitoringController` factory method checks on environment variables, you can set them in a test statically. Therefore, they are set before creating the first `MonitoringController`.

```
package example.probe.test;

import kieker.monitoring.core.configuration.ConfigurationKeys;
import org.junit.Test;

public class ExampleProbeTest {

        /**
         * Set system properties before instantiation anything.
         * Otherwise the MonitoringController will not see the
```

(continues on next page)

(continued from previous page)

```
         * configuration.
         */
        static {
                System.setProperty(ConfigurationKeys.CONTROLLER_NAME,
↪"ExampleProbeTest Controller");

                System.setProperty(ConfigurationKeys.WRITER_CLASSNAME,
                        TestDummyWriter.class.getCanonicalName());
        }

        @Test
        public void test() {
                final ExampleProbe probe = new ExampleProbe();
                probe.takeMeasurement("myOperation()", "example.ExampleClass");

                /** first record. */
                final IMonitoringRecord metadata = TestDummyWriter.getEvents().get(0);

                Assert.assertEquals("First record should be KiekerMetaData",
                        metadata.getClass().getName(),
                        KiekerMetadataRecord.class.getName());

                /** second record. */
                final IMonitoringRecord beforeEvent =
                        TestDummyWriter.getEvents().get(1);

                Assert.assertEquals("First record should be KiekerMetaData",
                        beforeEvent.getClass().getName(),

                BeforeOperationEvent.class.getName());
        }
}
```

In this test class, we set two properties. Firstly, we specify a controller name. This helps when debugging tests, as we can check whether the used controller is really the one with the internal name "ExampleProbeTest Controller". Secondly, we set the writer class. By default Kieker would write into a text log file. However, during testing we do not want that Kieker creates a directory and stores log information there. Instead we want to access logged data programmatically. The `TestDummyWriter` allows to access events from a statically defined internal list, which is most convenient for testing. The list is statically accessed with `TestDummyWriter.getEvents()`. The first event is always `KiekerMetadataRecord`, except you configure the controller to omit the metadata record.

Based on this simple setup, you can test your own probes easily. Please note, currently the `TestDummyWriter` is still part of iObserve and will move to Kieker in the near future.

1.3.14 How to use JMS Reader and Writer

This is a short introduction to the JMS reader and writer of **Kieker** named **AsyncJmsWriter** and **JmsReaderStage**. The directory `examples/userguide/appendix-JMS/` contains the sources, gradle scripts etc. used in this example. It is based on the Bookstore application with manual instrumentation presented **getting-started_**.

The following sections provide step-by-step instructions for the ActiveMQ JMS server implementation. The general procedure for this example is the following:

1. Download and prepare the respective JMS server implementation
2. Copy required libraries to the example directory

3. Start the JMS server
4. Start the analysis instance which receives records via JMS
5. Start the monitoring instance which sends records via JMS

Note: Due to a bug in some JMS servers, avoid paths including white spaces.

ActiveMQ

Download and Prepare ActiveMQ

Download an ActiveMQ archive from <http://activemq.apache.org/download.html> and decompress it to the base directory of the example. Note, that there are two different distributions, one for Unix/Linux/Cygwin and another one for Windows.

Under Unix-like systems, you need to set the executable-bit of the start script:

```
# chmod +x bin/activemq
```

Also under Unix-like systems, make sure that the file `bin/activemq` includes Unix line endings (e.g., using your favorite editor or the *dos2unix* tool).

Copy ActiveMQ Libraries

Copy the following files from the ActiveMQ release to the `lib/` directory of this example:

1. `activemq-all-<version>.jar` (from ActiveMQ's base directory)
2. `slf4j-log4j<version>.jar` (from ActiveMQ's `lib/optional` directory)
3. `log4j-<version>.jar` (from ActiveMQ's `lib/optional` directory)

Kieker Monitoring Configuration for ActiveMQ

The file `src-resources/META-INF/kieker.monitoring.properties-activeMQ` is already configured to use the **JmsWriter** via ActiveMQ. The important properties are the definition of the provider URL and the context factory:

```
kieker.monitoring.writer.jms.JmsWriter.ProviderUrl=tcp://127.0.0.1:61616/
kieker.monitoring.writer.jms.JmsWriter.ContextFactoryType=org.apache.activemq.jndi.
↪ActiveMQInitialContextFactory
```

Running the Example

The execution of the example is performed by the following three steps:

1. Start the JMS server (you may have to set your **JAVA_HOME** variable first): - `bin/activemq start` Start of the JMS server under UNIX-like systems - `bin/activemq start` Start of the JMS server under Windows
2. Start the analysis part (in a new terminal): - `./gradlew runAnalysisActiveMQ` Start the analysis part under UNIX-like systems - ‘‘gradlew.bat runAnalysisActiveMQ‘‘Start the analysis part under Windows

3. Start the instrumented Bookstore (in a new terminal): - `./gradlew runMonitoringActiveMQ` Start the analysis part under UNIX-like systems - `gradlew.bat runMonitoringActiveMQ` Start the analysis part under Windows

1.3.15 How to use AMQP Writer and Reader

This chapter gives a brief description on how to use the **AmqpWriter** and **AMQPReaderStage** classes, which allow to use Kieker with AMQP-based queue implementations such as *RabbitMQ <http://www.rabbitmq.com>*. The directory `examples/userguide/appendix-AMQP/` contains the sources, gradle scripts and other sources used in this example. It is based on the Bookstore application.

The following paragraphs provide step-by-step instructions for the popular AMQP implementation RabbitMQ.

Preparation

Download and Install RabbitMQ

Download the RabbitMQ distribution from http://www.rabbitmq.com/download.html and follow the installation instructions for your OS. Since RabbitMQ requires Erlang, additional software packages may have to be installed on your machine.

In order to use RabbitMQ's integrated management UI, you may have to enable the appropriate plugin first. This is done by issuing the following command from the command line.

- `rabbitmqplugins enable rabbitmq management` Enable the management UI under UNIX-like systems
- `rabbitmqplugins enable rabbitmq management` Enable the management UI under Windows]

Once the UI is enabled, you may access it at port 15672 by default.

Configure RabbitMQ

Once the RabbitMQ server is installed and started, create a queue for Kieker to use. This can be done easily using RabbitMQ's management UI. It is accessible via http://localhost:15672 (the default credentials are *guest:guest*) We will assume a queue named `kieker` for the remainder of this example. Please note the following caveats when configuring the server:

1. If you choose to create a transient queue, the entire queue (not just the queued messages) is destroyed on server shutdown and must be re-created manually.
2. The RabbitMQ server's default permissions grant access only from `localhost`. If your RabbitMQ server runs on a remote machine, you have to set the permissions accordingly.

Kieker Monitoring Configuration for RabbitMQ

The file `src-resources/META-INF/kieker.monitoring.properties` is already configured to use the **AmqpWriter**. The important properties are the server URI and the queue name.

```
kieker.monitoring.writer.amqp.AmqpWriter.uri=amqp://guest:guest@127.0.0.1
kieker.monitoring.writer.amqp.AmqpWriter.queuename=kieker
```

Running the Example

The execution of the example is performed by the following three steps:

1. Ensure that the RabbitMQ server is started and the configured queue is accessible.
2. Start the analysis part (in a new terminal): - `# ./gradlew runAnalysisAMQP` Start the analysis part under UNIX-like systems - `# gradlew.bat runAnalysisAMQP` Start the analysis part under Windows]
3. Start the instrumented Bookstore (in a new terminal): - `# ./gradlew runMonitoringAMQP` Start the analysis part under UNIX-like systems - `# gradlew.bat runMonitoringAMQP` Start the analysis part under Windows

1.4 Instrumenting Software

Kieker allows to instrument various types of applications and services utilizing different techniques to instrument and implement probes. Yet the monitoring data produced can be analyzed by all **Kieker** tools.

Kieker supports a ever growing variety of programming languages and technologies to measure runtime information of your software systems. In general **Kieker** uses probes to collect information which are then send to a logging facility. To introduce the probes into your software system, **Kieker** uses different techniques including aspect-oriented programming. They allow the introduction of probes without changing the source code. For rare cases, where no such technique is applicable, **Kieker** can be introduced manually.

1.4.1 Instrumenting Java

This section comprises information on instrumenting Java application including creating and using probes, writers, samplers and related topics.

Configuring Kieker

- Kieker uses a configuration file *kieker.monitoring.properties*
- Must be placed depending on application type (see related)
- Different options to log data (refer to real info where by crosslink)

Todo: Add the following info to kieker configuration.

- CSV logging file (storage on the monitoring system)
- Binary logging file (storage on the monitoring system)
- Compressed logging file (binary and CSV, storage on the monitoring system)
- TCP binary stream (transfer to remote host)
  - Kieker binary transport protocol (two TCP connections supporting a prioritized second channel)
  - ExploreViz binary transport protocol (single TCP connection)
  - Modern Kieker binary transport protocol (experimental)
- JMS object message transport (transfer to remote host)
- JMX transport via notifications (transfer to remote host)
- UNIX based named pipes (local transfer)

- Database writer (experimental)
- AMQP protocol message support (transfer to remote host)

Kieker Monitoring instances can be configured by properties files, **Configuration** objects, and by passing property values as JVM arguments. If no configuration is specified, a default configuration is used. The default configuration can be found here including documentation for all properties. Additional information can be found within the documentation of the **Monitoring Controller**, **Monitoring Probes** and **Monitoring Writers**. The default configuration properties file, which can be used as a template for custom configurations, is provided by the file `kieker.monitoring.example.properties` in the directory `examples/` directory of the binary release.

Configurations for Singleton Instances

In order to use a custom configuration file, its location needs to be passed to the JVM using the parameter *kieker.monitoring.configuration* as follows:

```
java -Dkieker.monitoring.configuration=<ANY-DIR>/my.kieker.monitoring.properties [...]
```

Alternatively, a file named `kieker.monitoring.properties` can be placed in a directory called `META-INF/` located in the classpath. The available configuration properties can also be passed as JVM arguments, e.g., `-Dkieker.monitoring.enabled=true`.

Configurations for Non-Singleton Instances

The class **Configuration** provides factory methods to create **Configuration** objects according to the default configuration or loaded from a specified properties file: `createDefaultConfiguration`, `createConfigurationFromFile`, and `createSingletonConfiguration`. Note, that JVM parameters are only evaluated when using the factory method `createSingletonConfiguration`. The returned `Configuration` objects can be adjusted by setting single property values using the method `setProperty`.

Manual Instrumentation

Manual instrumentation is usually not the right way to instrument larger applications. However, to inspect smaller portions of an application in an ad-hoc manner or in cases aspect-weaving is not possible, manual instrumentation can be a viable option.

To use Kieker with an Java application, you have to add the dependency to your build system, e.g., (in gradle)

```
compile 'net.kieker-monitoring:kieker:1.14'
```

See also https://mvnrepository.com/artifact/net.kieker-monitoring/kieker

Instrumentation requires three key elements: - A `MonitoringController` - Data collection - Logging of the data

Monitoring Controller

The `MonitoringController` provides basic facilities for monitoring and logging, including a source for timestamps. It can be obtained in any class by

```
private static final IMonitoringController MONITORING_CONTROLLER =
                MonitoringController.getInstance();
```

This returns a singleton instance of the monitoring controller.

Data Collection

Usually in data collection you gather all information you want to store and put that data into an instance of an event type.

```
final long tin = MONITORING_CONTROLLER.getTimeSource().getTime();
final String operationName = "public void exampleOp()"
final String className = this.getClass().getName();
```

In this example, the first line uses the time source facility of the `MonitoringController` to gain the current time. Subsequent, two strings are defined which represent the name of the operation (method) and the name of the class the method resides in. Finally, the data must be packed into a event type.

```
final BeforeOperationEvent e =
     new BeforeOperationEvent(tin, className, operationName);
```

Logging of Data

The last step uses the logging facility of the `MonitoringController`.

Instrumentation with AspectJ

Todo: Describe general approach to do so. Refer to a tutorial for a quicker approach.

The **tutorial-servlet-example_** contains a some basic introduction to using AspectJ probes.

AspectJ Configuration

Compile-time weaving

Load-time weaving

References

- AspectJ probes

Servlet Instrumentation

Servlets can be instrumented utilizing `javax.servlet.ServletContextListerner`, `javax.servlet.http.HttpSessionListener` and `javax.servlet.Filter`.

The first is triggered when a Servlet context is created (instantiation of the Servlet) and destroyed. The second is triggered every time a new session is created. And the last is invoked every time a request is sent to the Servlet. In the following we will address all three types.

Please note that Servlets can also use other listeners which could in principle also used to trigger monitoring. However, such probes do not exist within **Kieker**, but can be build easily with **Kieker** framework functionality..

Servlet Context Listener

Todo: Add how to add context listeners here (iObserve)

HTTP Session Listener

Todo: Add how to add session listeners here (Kieker, iObserve)

Servlet Filter

The Java Servlet API includes the `javax.servlet.Filter` and interface. It can be used to implement interceptors for incoming HTTP requests. **Kieker** uses this interface to implement different probes. To add such interceptor to a Servlet, you have to edit the `web.xml` file in your Servlet project. For example:

```
<filter>
        <filtername>sessionAndTraceRegistrationFilter</filtername>
        <filterclass>kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter
↪</filterclass>
        <initparam>
                <paramname>logFilterExecution</paramname>
                <paramvalue>true</paramvalue>
        </initparam>
</filter>
<filtermapping>
        <filtername>sessionAndTraceRegistrationFilter</filtername>
        <urlpattern>/</urlpattern>
</filtermapping>
```

This configuration adds the `kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter` interceptor to the Servlet configuration and identifies it with `sessionAndTraceRegistrationFilter`. It sets one parameter `logFilterExecution` to `true`. In the filter mapping, the `sessionAndTraceRegistrationFilter` is mapped to all URLs, i.e., to all Servlet in the project.

Related Information

Kieker comes with many different Servlet filters.

Todo: Add list to filters and listeners here

Instrumentation with CXF Interceptors

Instrumentation with DiSL

https://gitlab.ow2.org/disl/disl

Instrumentation with AIM

Todo: WE need to add documentation here.

Instrumentation of Java EE Applications

Instrumentation of Spring Applications

1.4.2 Instrumenting C and other Native Programming Languages

We provide experimental C language support for Kieker.

Creating your own Event Types

Instrumentation

- Include instrumentation with gcc feature
- Using AspectC++ for instrumentation

1.4.3 Instrumenting Perl

Perl (experimental, http://eprints.uni-kiel.de/21141/7/vortrag.pdf)

- Sub::WrapPackages based AOP
- Manual instrumentation

Note: The code generator for Kieker records can produce record types for Perl.

1.4.4 Python Instrumentation

Note: This is an upcoming effort. Currently, we are selecting a model to create fast and sufficient record types in python.

1.4.5 Kieker4COM

Kieker4COM is a Kieker adapter supporting monitoring of programming languages based on Microsoft's Component Object Model (COM). The adapter has been developed as a part of the DynaMod research project. It has been tested particularly with Visual Basic 6.

Downloading, Installing, Using Kieker4COM

Downloading Kieker4COM

- Kieker4COM install archives are provided by the nightly build
- The sources are available via the Git repository kieker4com

Installing Kieker4COM

Note: If you have just uninstalled a Kieker4COM version, you should perform a restart before starting a new installation process!

1. Start installer

Double-click on the downloaded file to start the installer.

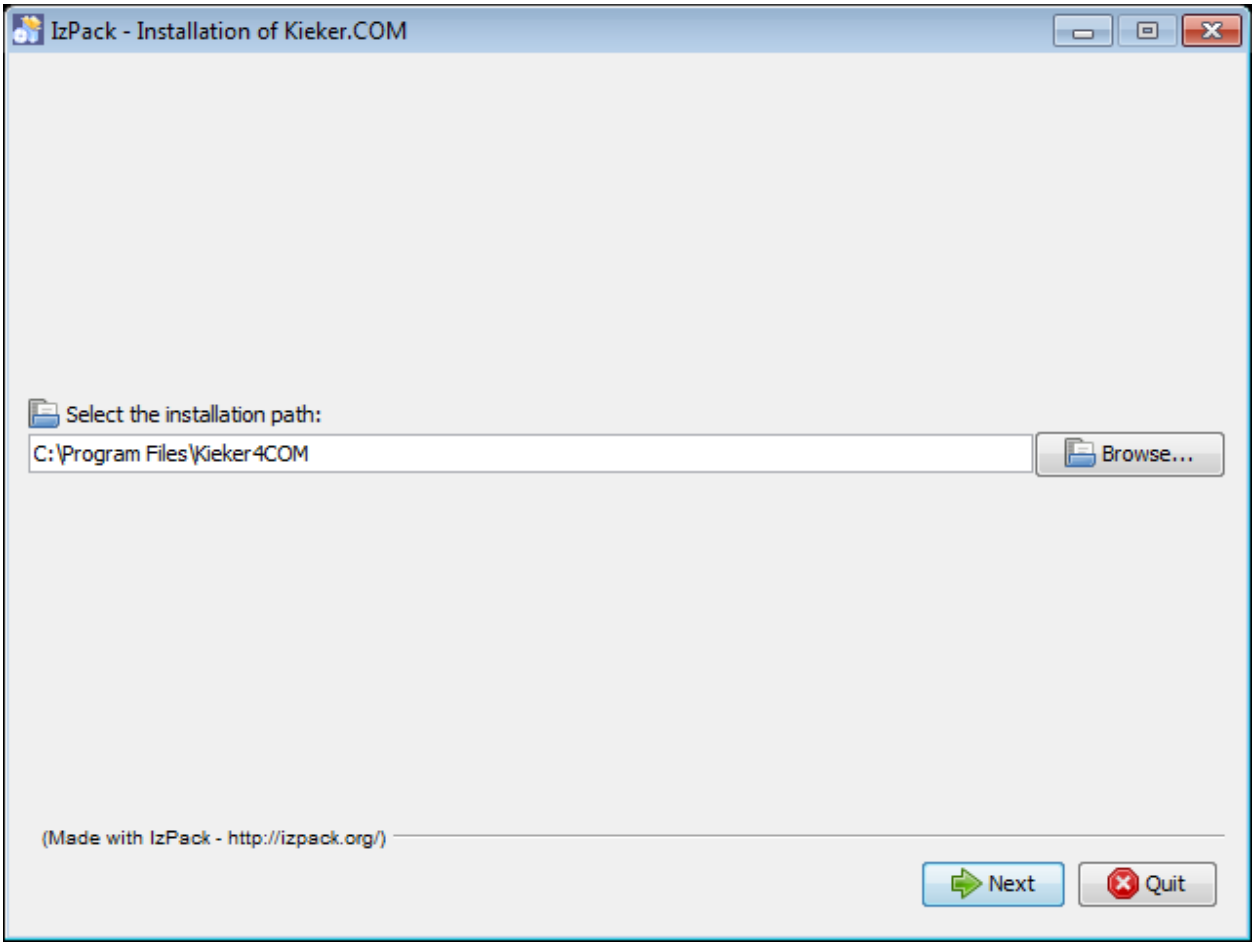
You may need to allow the execution of the Java Virtual Machine required for the installation program.

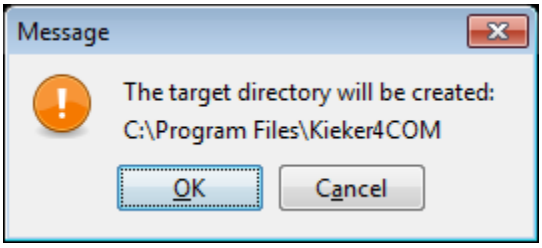
2. Language Selection

In the following dialog, you can select the language used in the installation wizard. Currently, *German* and *English* are supported.

3. Installation Directory

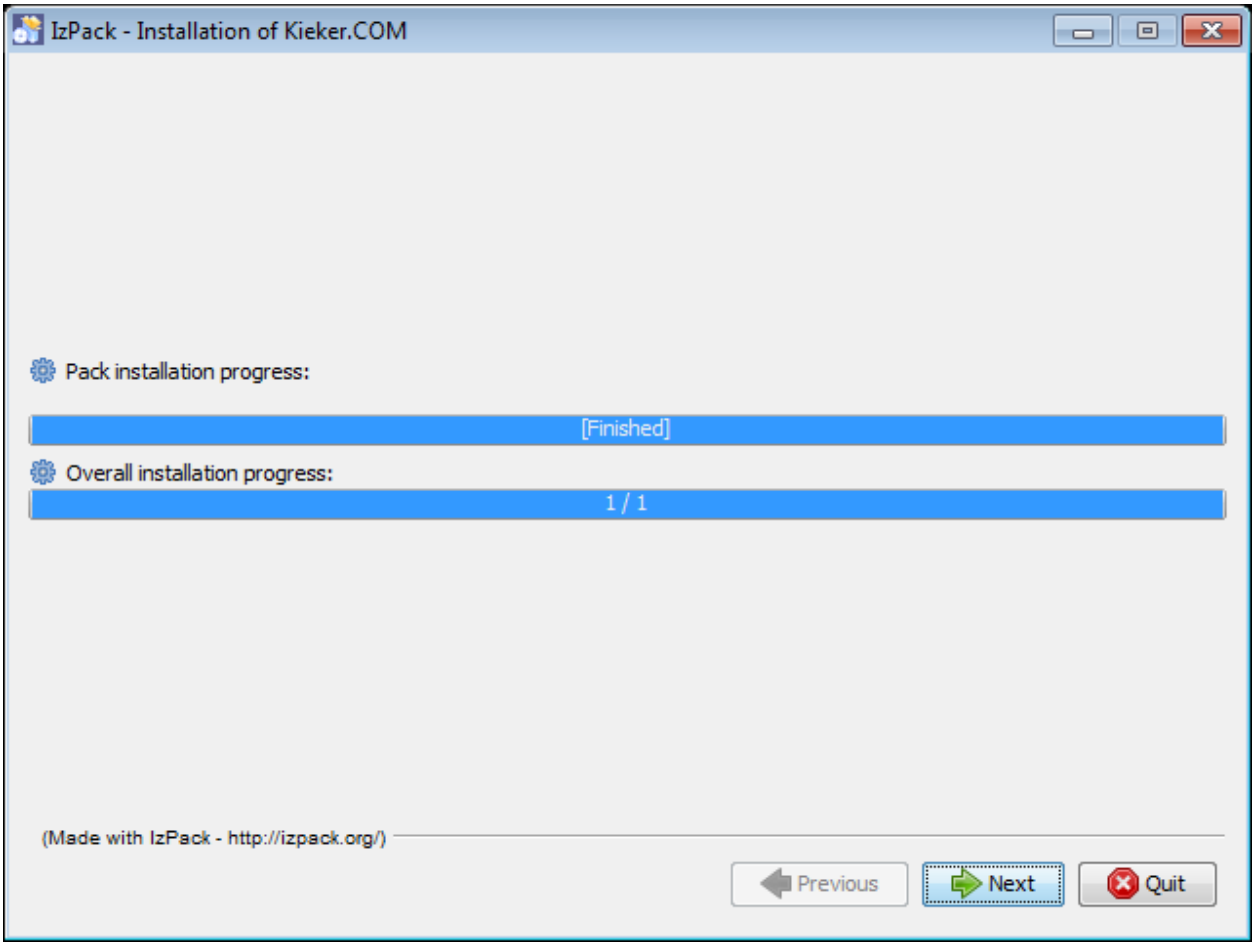
The next step of the installation wizard lets you select the Kieker4COM installation directory. Currently, our recommendation is to keep the default value. In the following step, this step of creating the installation directory requires an additional confirmation.





4. Installation of Kieker4COM Binaries

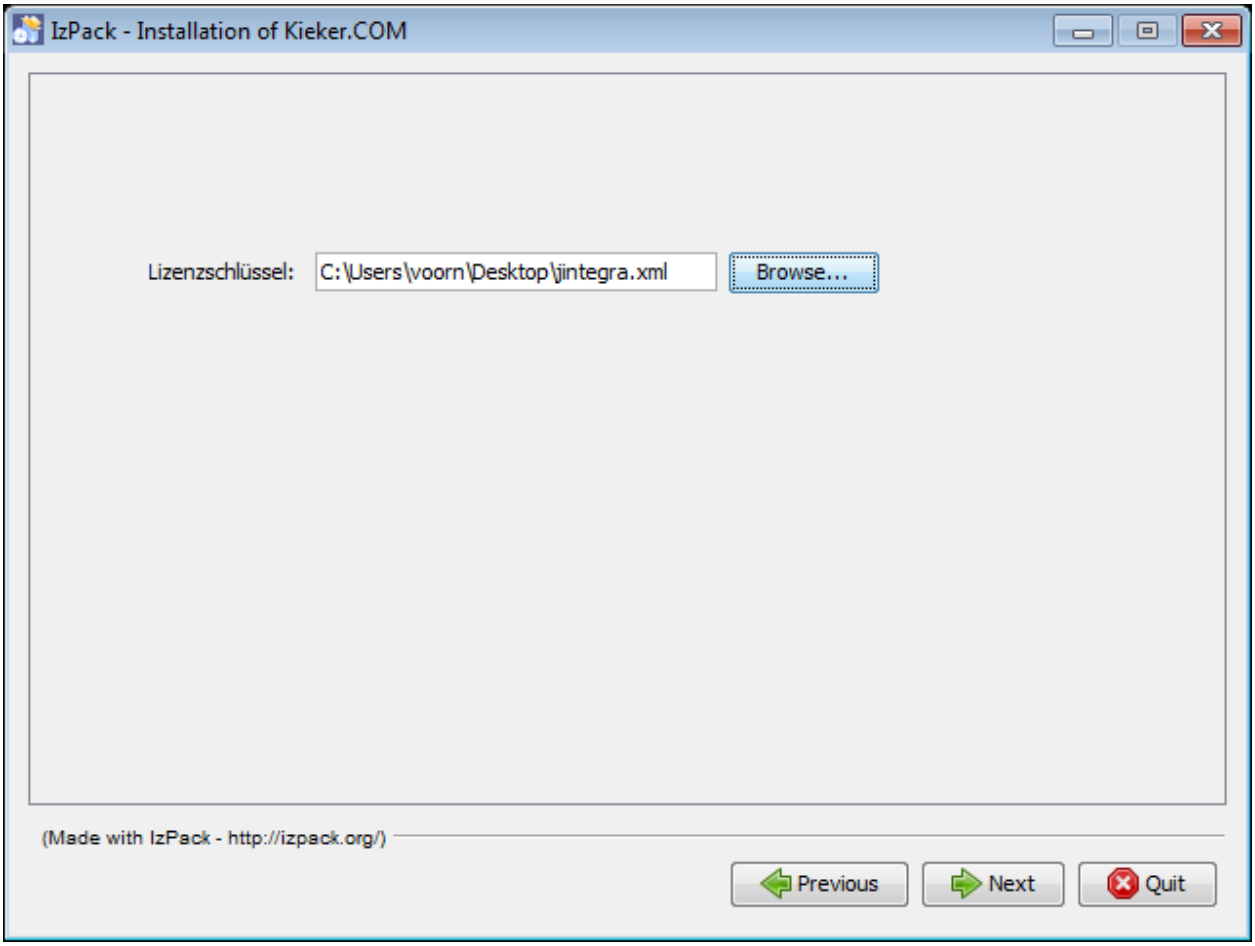
After having confirmed the installation in the previous step, the installer copies the Kieker4COM binaries to the selected directory.



5. Selection of J-Integra COM License File

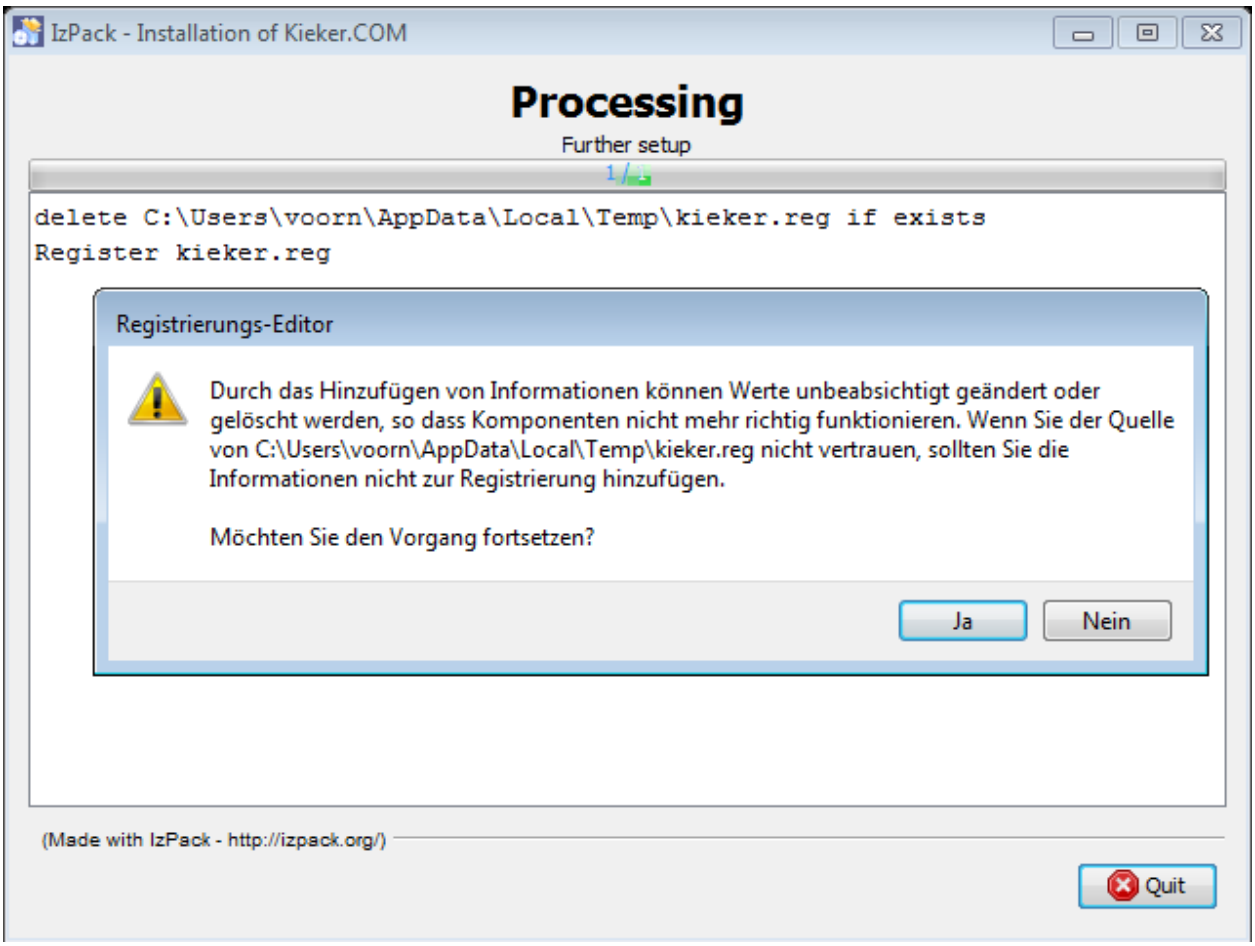
Kieker4COM employs the ‘ J-Integra COM <http://j-integra.intrinsyc.com/com.asp>‘__ bridge for accessing the Java-based Kieker monitoring component. The use of J-Integra COM requires the installation of a ‘ JI COM Client license <http://j-integra.intrinsyc.com/pricing.asp>‘__.

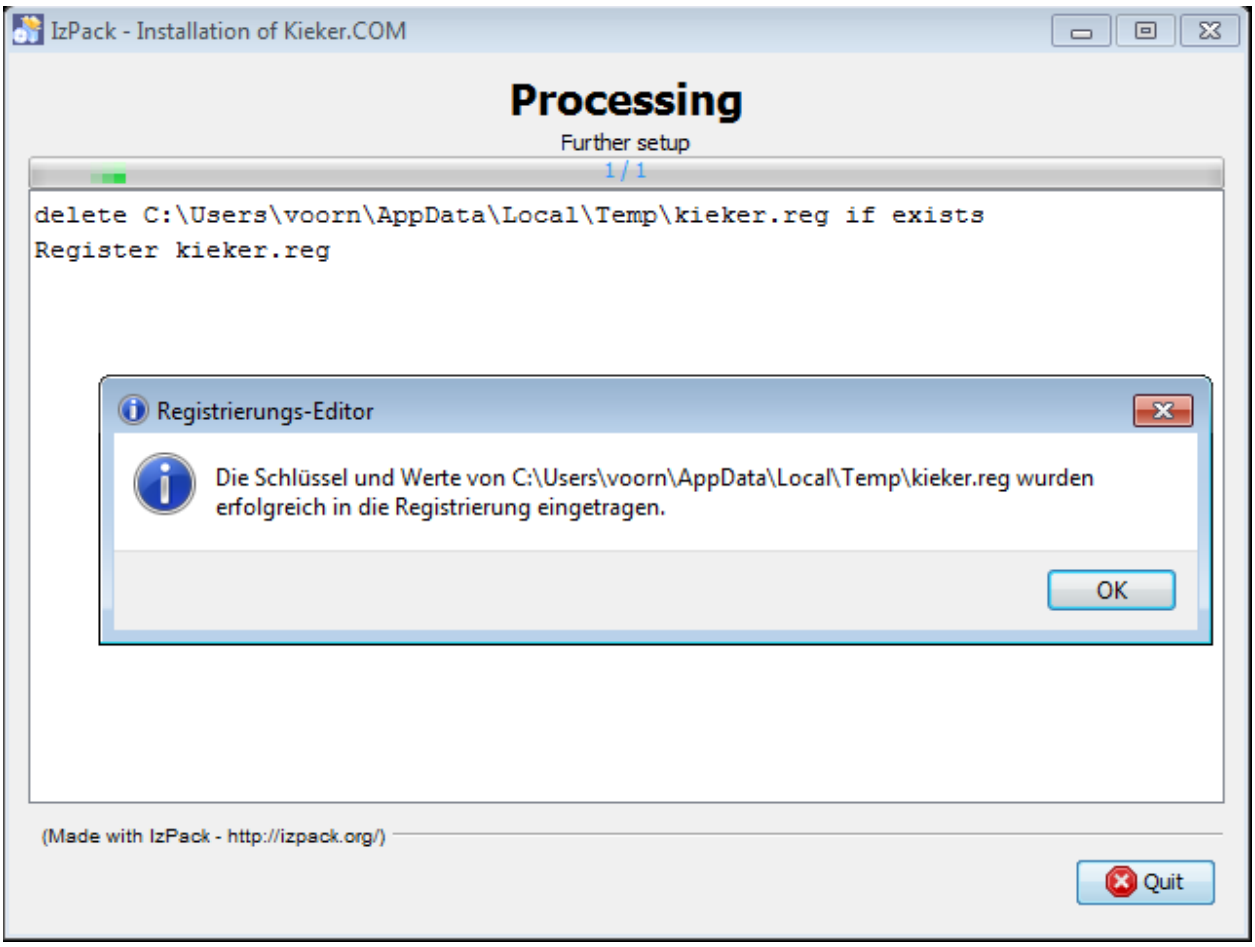
Please select the file system location of the J-Integra COM license file and confirm your selection.

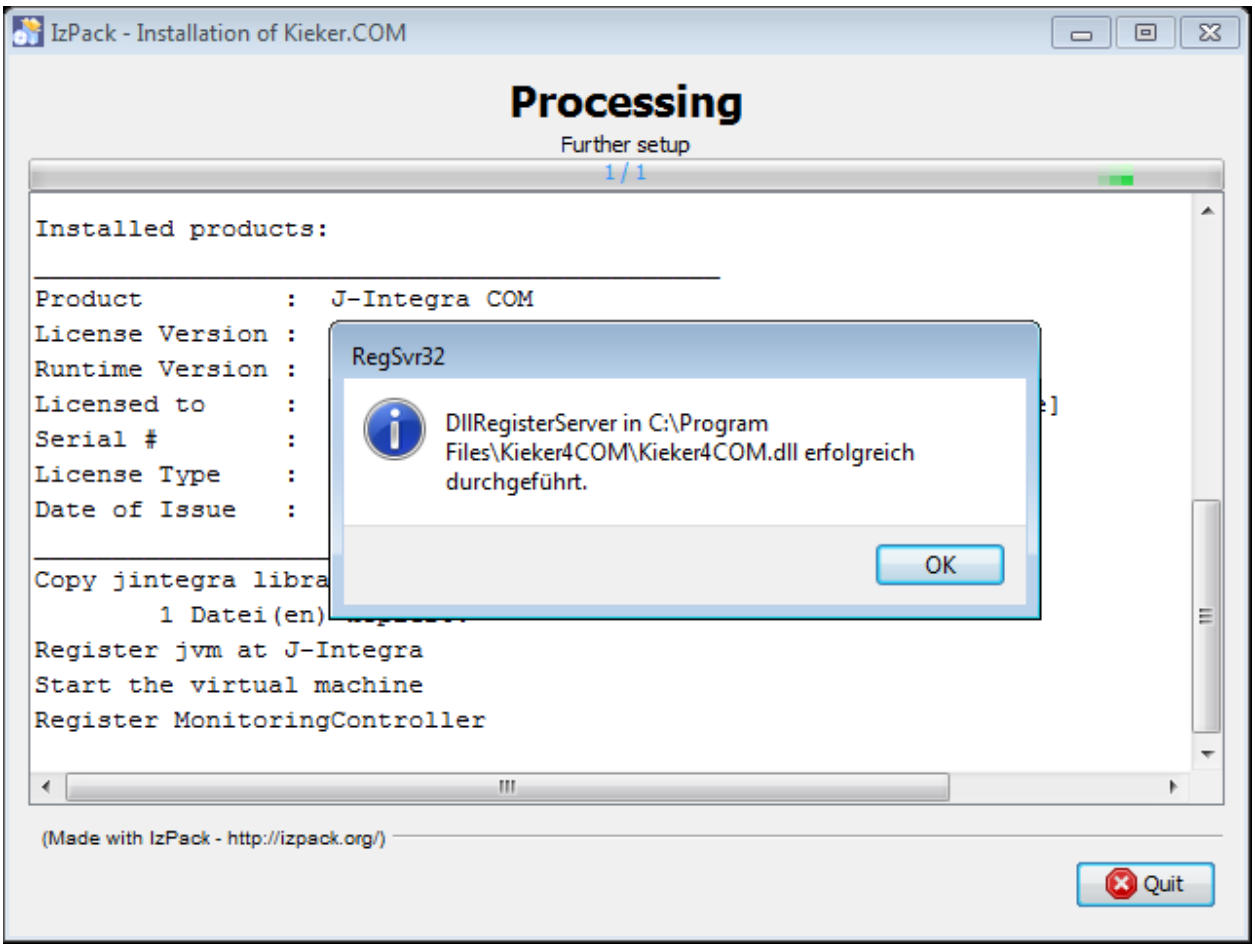


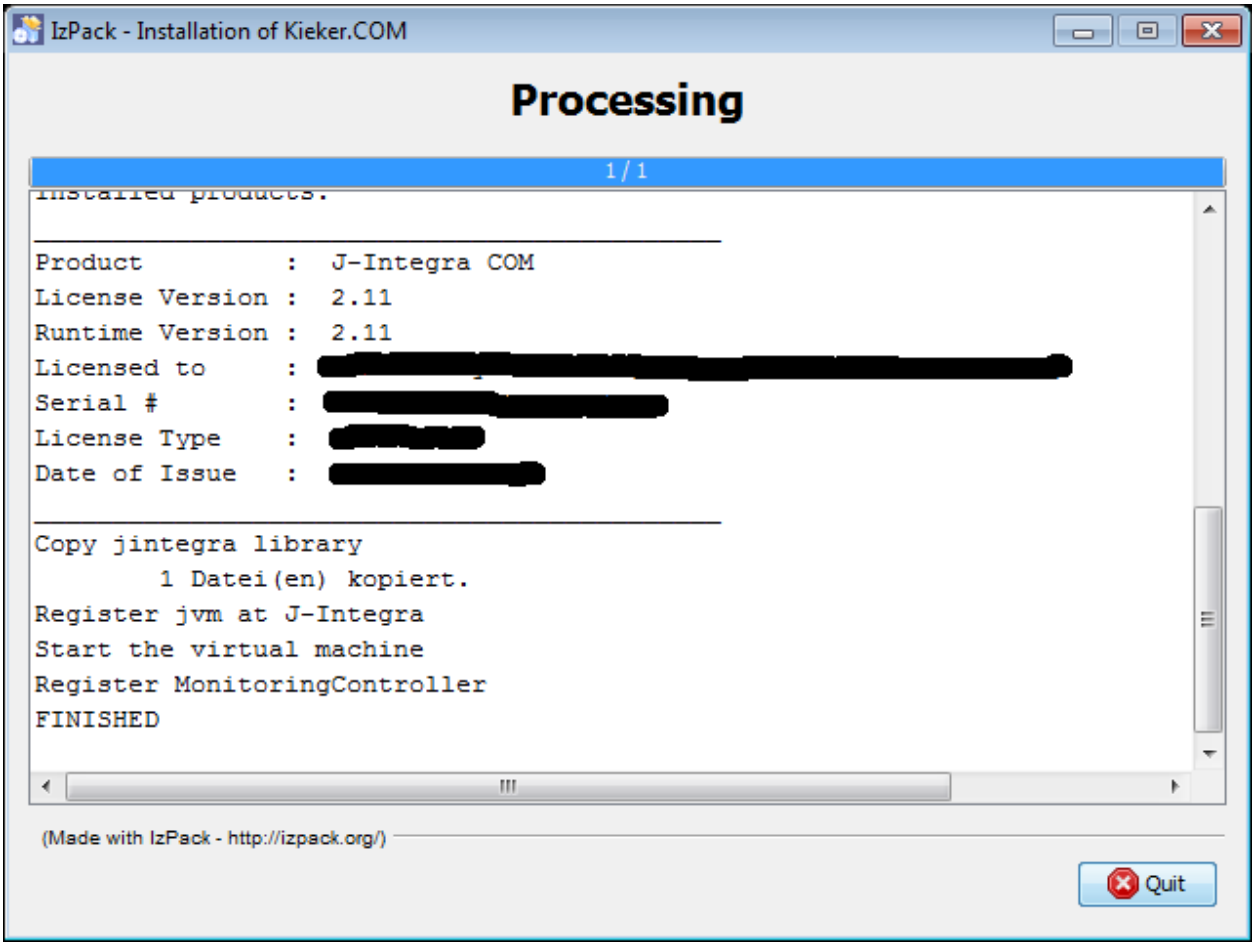
6. Registration of Kieker4COM and Completion of Installation

The next installation steps include the registration of the Kieker4COM service in the Windows registry, the activation of the J-Integra COM installation included with Kieker4COM, as well as an initial start of the Kieker4COM service.


IzPack - Installation of Kieker.COM
Processing
Further setup
1 / 1
delete C:\Users\voorn\AppData\Local\Temp\kieker.reg if exists
Register kieker.reg
Registrierungs-Editor
Durch das Hinzufügen von Informationen können Werte unbeabsichtigt geändert oder gelöscht werden, so dass Komponenten nicht mehr richtig funktionieren. Wenn Sie der Quelle von C:\Users\voorn\AppData\Local\Temp\kieker.reg nicht vertrauen, sollten Sie die Informationen nicht zur Registrierung hinzufügen.
Möchten Sie den Vorgang fortsetzen?
Ja
Nein
(Made with IzPack - http://izpack.org/)
Quit


IzPack - Installation of Kieker.COM
Processing
Further setup
1 / 1
delete C:\Users\voorn\AppData\Local\Temp\kieker.reg if exists
Register kieker.reg
Registrierungs-Editor
Die Schlüssel und Werte von C:\Users\voorn\AppData\Local\Temp\kieker.reg wurden erfolgreich in die Registrierung eingetragen.
OK
(Made with IzPack - http://izpack.org/)
Quit


IzPack - Installation of Kieker.COM
Processing
Further setup
1 / 1
Installed products:
Product : J-Integra COM
License Version :
Runtime Version :
Licensed to :
Serial # :
License Type :
Date of Issue :
Copy jintegra libra
1 Datei(en)
Register jvm at J-Integra
Start the virtual machine
Register MonitoringController
RegSvr32
DllRegisterServer in C:\Program Files\Kieker4COM\Kieker4COM.dll erfolgreich durchgeführt.
OK
(Made with IzPack - http://izpack.org/)
Quit

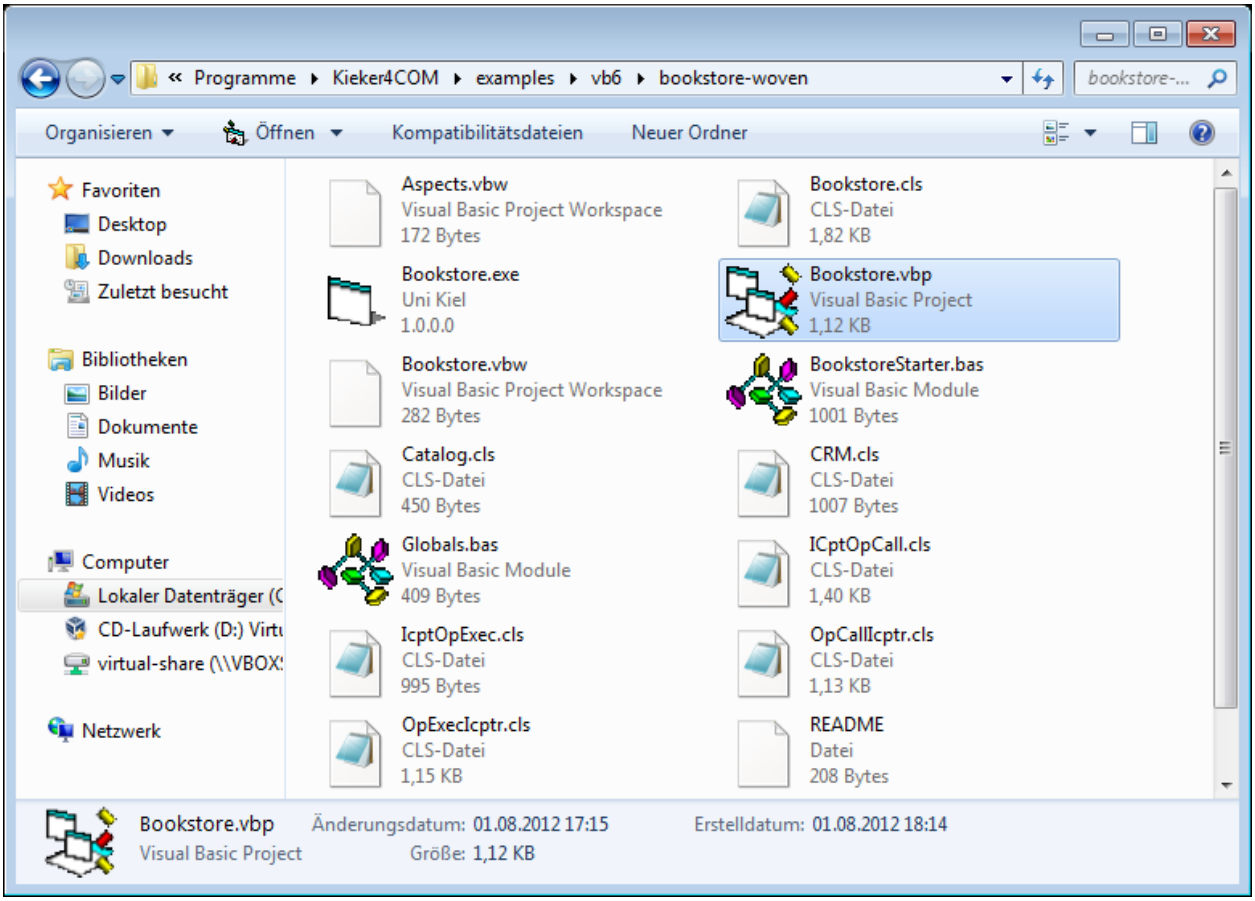


Testing the Kieker4COM installation

The Kieker installation directory (%KIEKER_HOME%) contains a folder called examples, which includes example projects instrumented in different programming langugages. The directory examplesvb6\ includes examples for Visual Basic 6:

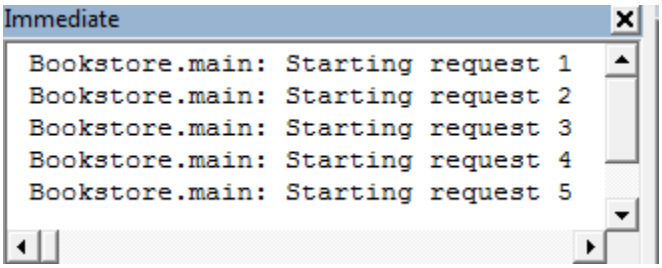
1. bookstore-annotated. A sample application which is enriched by AspectVB6 monitoring annotations which can be processed by the ‘ AspectLegacy <http://build.se.informatik.uni-kiel.de/DynaMod-tools/trac/>‘__ tool in order to weave Kieker4COM monitoring aspects into the VB6 source code. See the Wiki page Kieker-COM/Aspects for details.
2. bookstore-woven. This project is the result of the afore-mentioned process of weaving Kieker4COM monitoring aspects into the source code of the bookstore-annotated project. We will use this project to test the Kieker4COM installation.

The following figure shows the directory contents:

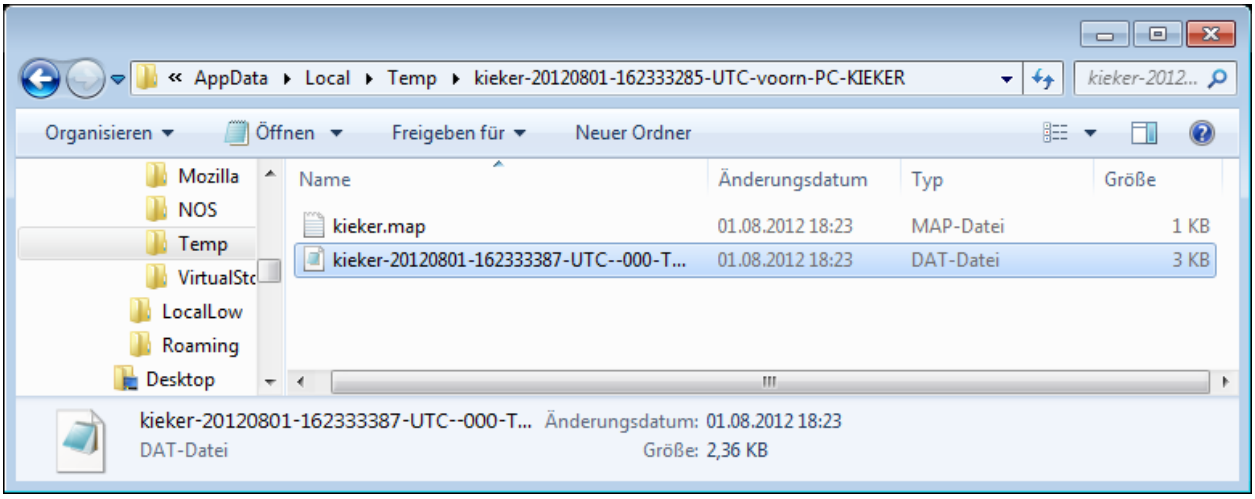


UPDATE: In newer versions, the example directory includes a pre-compiled Bookstore.exe which can be started directly without the need to import the VB6 project.

Import the project into the Visual Basic 6 IDE by opening the project file Bookstore.vbp. Having started the example, the following debug messages should appear in the *Immediate Window* (Ctrl+G):

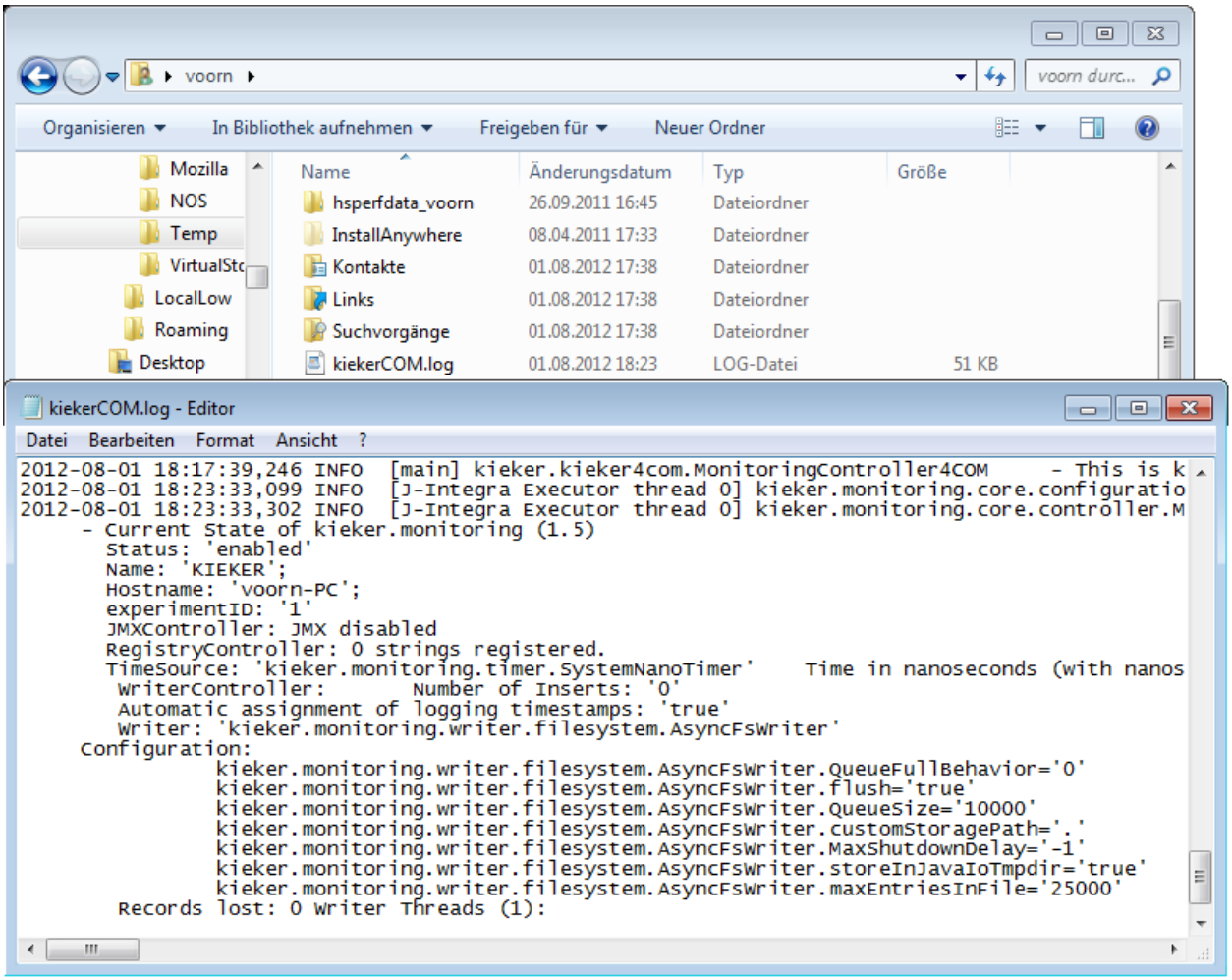


The Kieker monitoring log is written to a directory named like kieker-<timestamp> located in the %TEMP% directory (e.g., *C:UsersvoornAppDataLocalTemp*).



This Kieker file system monitoring log can now be processed by the Kieker.TraceAnalysis tool, just like monitoring logs from Java or .NET systems. An example monitoring log is contained in the examplesvb6monitoring-logsdirectory.

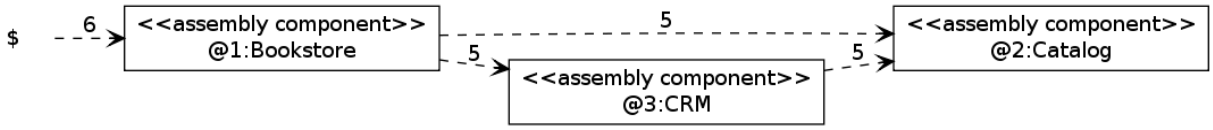
A KiekerCOM.log file with log messages is written to the %USERPROFILE% directory.

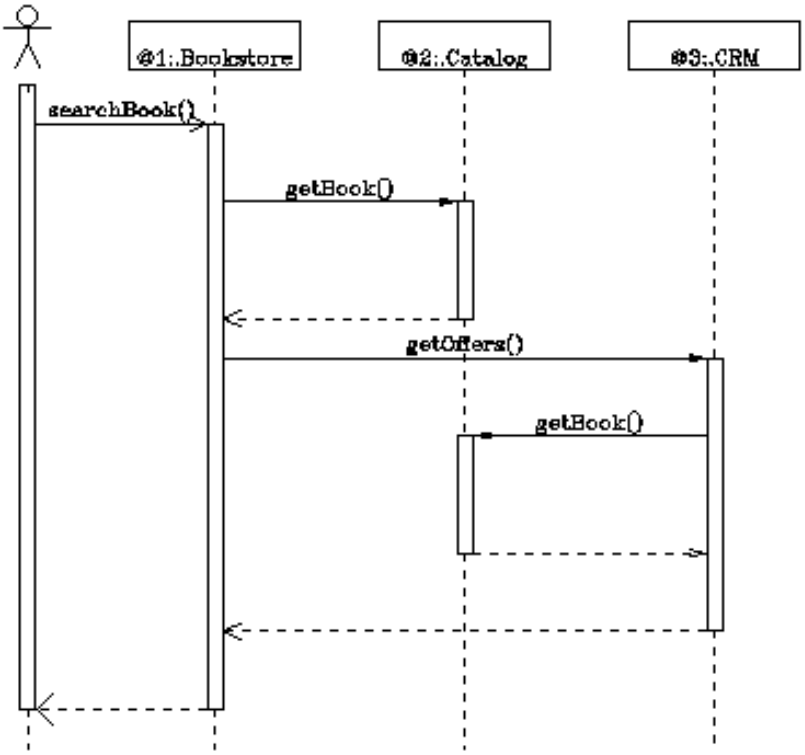


The following diagrams were created by the following calls to the Kieker.TraceAnalysis tool:

C:Program Fileskieker4COMbin>trace-analysis.bat -i ..examplesvb6monitoring-logskieker-20111017-152928614-

UTC-voorn-PC-KIEKER -o %TEMP% -p bla –plot-Assembly-Component-Dependency-Graph –plot-Assembly-Component-Dependency-Graph





Log messages are written to a kieker.log file in the %USERPROFILE% directory.

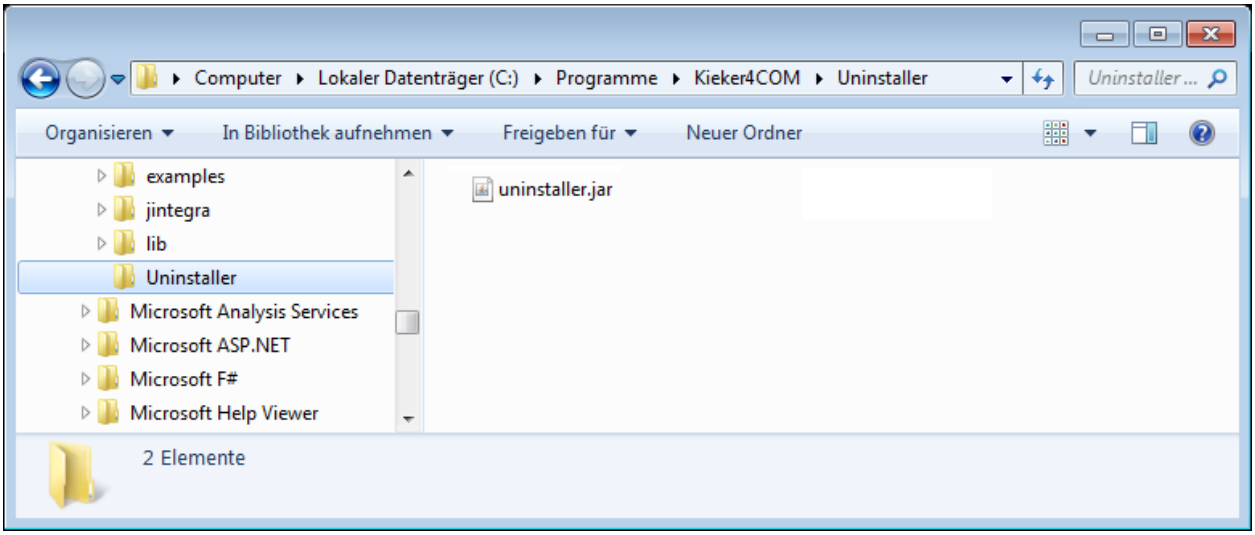
Please refer to the ‘ Kieker documentation <https://se.informatik.uni-kiel.de/kieker/documentation/>‘__—particularly the User Guide—to learn more about the usage of the Kieker.TraceAnalysis tool.

Uninstalling Kieker4COM

1. Start Uninstaller

Double-click on the *uninstaller.jar* file, to be found in the *Uninstaller* sub-directory.

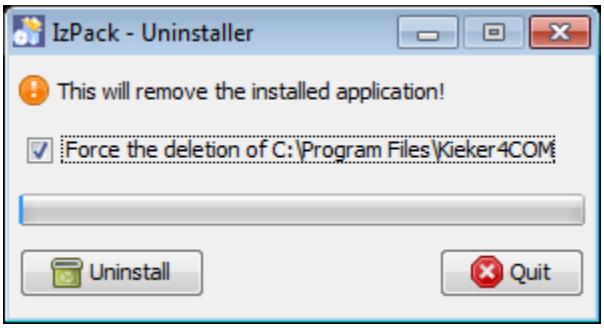
You may need to allow the execution of the ‘ Java Virtual Machine <http://en.wikipedia.org/wiki/Java_Virtual_Machine>‘__ required for the uninstaller.

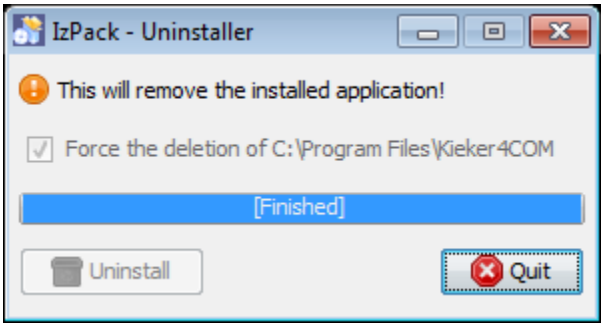


2. Confirm Uninstallation

In the upcoming wizard you should select the deletion of all files included in the Kieker4COM installation directory and start the uninstall process.

The uninstaller reports the successful deregistration of the Kieker4COM service and the successfull completion of the uninstallation process.





3. Manual Deletion of the Kieker4COM Installation Directory

The uninstaller already removed most of the sub-directories and files included in the Kieker4COM installation directory. As a last step, you'll need to manually remove the kieker4COM directory from your %ProgramFiles% (e.g., C:Programmekieker4COM) directory.

In some cases, the file *Kieker4COM* cannot be removed because it used. Please perform a restart an repeat this manual deletion step.

Important note for Subsequent Reinstallation

You should restart your system after an uninstallation before starting a subsequent installation.

Kieker4COM Aspects

Please see *Getting Started* to learn how to install and use Kieker4COM. The pathes mentioned in this document refer to the installation directory.

Kieker4COM VB6 Aspects Project

The Kieker4COM aspects project directory for VB6 can be found in the directory Kieker4COMaspectsvb6. The VB6 project file, which can be imported into the Visual Basic 6 IDE and can be used with Aspect VB6, is *Kieker4COMaspectsvb6Aspects.vbp*.

Todo: The tools site has moved. Please fix it.

Currently, the project includes two aspects for monitoring executions (OpExecIcptr) and calls (OpCallIcptr) of VB6 routines, i.e., Procedures, Functions, and Properties:

1. OpExecIcptr.cls
2. OpCallIcptr.cls

Using the Kieker4COM Aspects

The directory Kieker4COMexamplesvb6bookstore-annotated contains a VB6 version of the Bookstore application, including annotations for the Kieker4COM aspects. These annotations can be processed by Aspect VB6.

Todo: The tools site has moved. Please fix it.

Adding Annotations to VB6 Source Code

The examples were taken from the KiekerCOM example project *Kieker4COMexamplesvb6bookstore-annotated*.

OpExecIcptr

```
'@intercept#Execution:OpExecIcptr["Bookstore","Class_Initialize"]

Private Sub Class_Initialize()

Set oCatalog = New catalog

...
```

OpCallIcptr

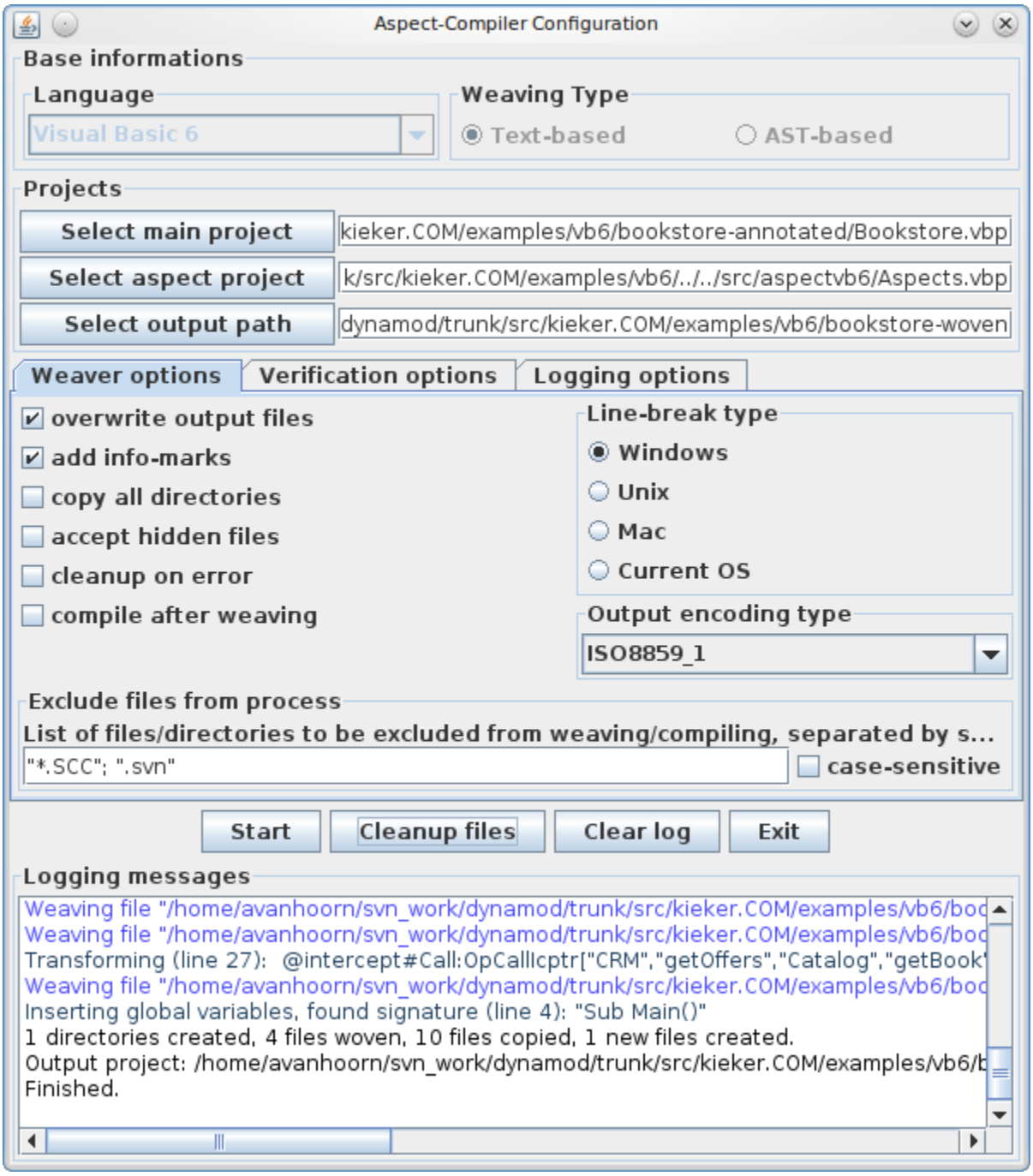
```
Public Sub searchBook()

'@intercept#Call:OpCallIcptr["Bookstore", "searchBook", "Catalog", "getBook"]

catalog.getBook (False)

'@intercept#Call:OpCallIcptr["Bookstore", "searchBook", "CRM", "getOffers"]

crm().getOffers

End Sub
```

Using AspectVB6 for Weaving the Monitoring Code

Using the command-line

```
/path/to/avb6c.sh \
  -s bookstore-annotated/Bookstore.vbp \
  -a ../../src/aspectvb6/Aspects.vbp -o bookstore-woven/
```

Using the GUI



1.4.6 Instrumenting Visual Basic 6

AspectLegacy Quick Start (Visual Basic 6)

Table of Contents

This section describes the steps to be done for installing the AspectLegacy tool. Note that the tool works under Linux as well as under Windows XP/Vista/7, but for compiling any (woven) Visual Basic 6 source code projects (.vbp), an installed version of the Visual Basic 6 integrated development environment (VB6 IDE) from Microsoft is required; since the IDE is only available for Windows machines, VB6 code cannot be compiled (but woven) under Linux.

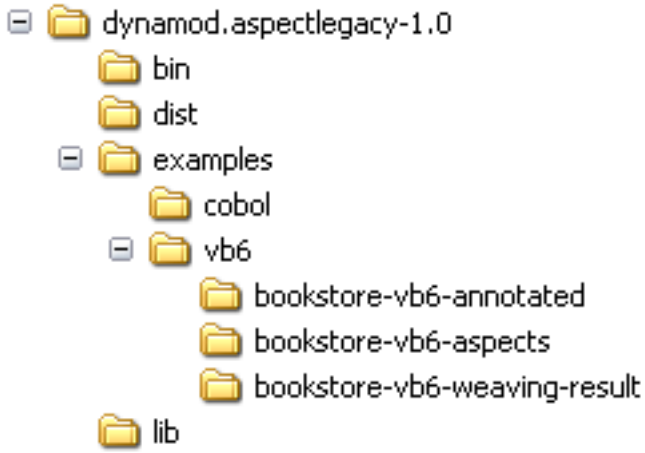
Introduction to the “Bookstore” Example

The AspectLegacy distribution contains a simple “bookstore” example code, written in Visual Basic 6 and divided into a main- and an aspects-project. Depending on this example, the section below describes how VB6 source files can be woven using the AspectLegacy tool, and how the woven code can be compiled afterwards. The source code of the bookstore main project contains several annotations referring to the classes of the aspects projects. They will take effect after weaving, by showing up dialogue windows whenever any of the annotated code positions are being entered while runtime.

Installation

It is assumed that you have already installed a VB6 IDE under Windows.

- At first, you have to download the AspectLegacy binary distribution archive (“dynamod.aspectlegacy-1.0_binaries”), which is available here as a .zip- as well as a .tar.gz-file.
- Extract the downloaded archive to an arbitrary location; the content will be placed into a (sub-)directory “dynamod.aspectlegacy-1.0” containing the following elements:



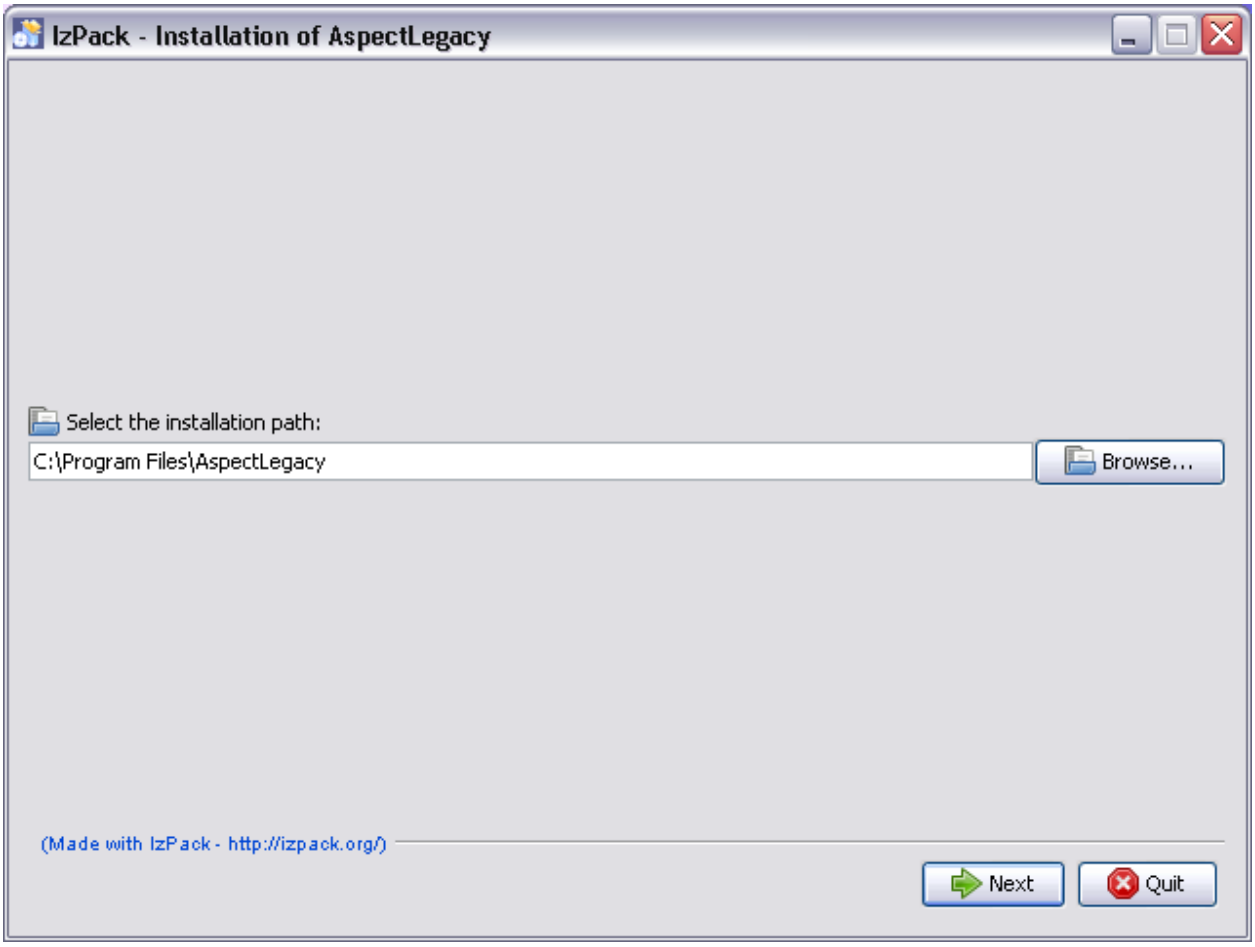
The content of the sub-folders is as follows:

Folder	Content
bin	Binary files (script files)
dist	Distribution files, contains the installer for the weaver.
examples	Examples for Cobol (futural feature) and Visual Basic 6.
lib	Required libraries.

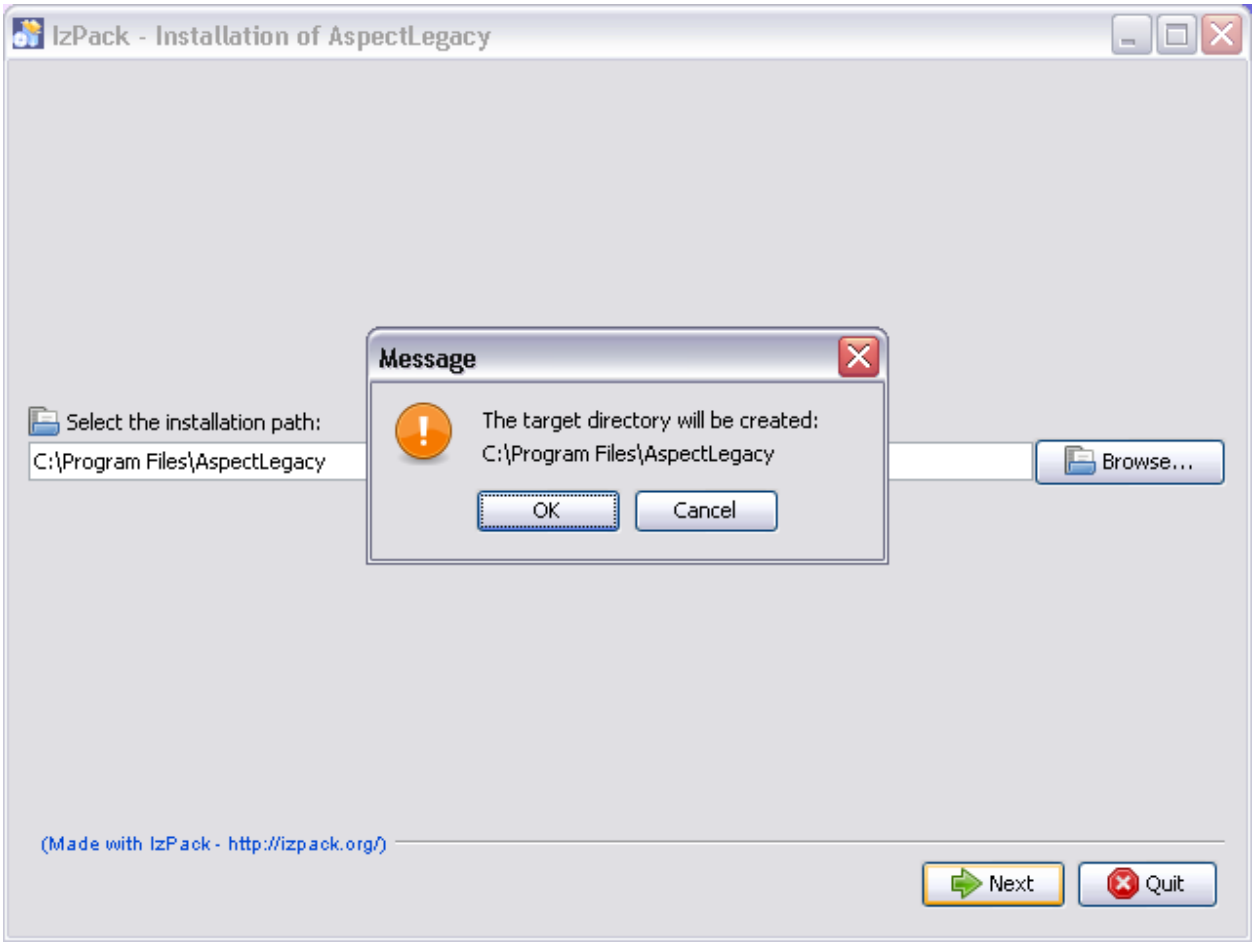
- Start the installer by double-clicking the “dynamod.aspectlegacy-1.0-installer-WIN32.jar” file, which is located in the “dist”-directory.
- Select the language to be used for the installation process:

Note that the language only affects the installation process, not the aspect legacy application itself, which is always in english.

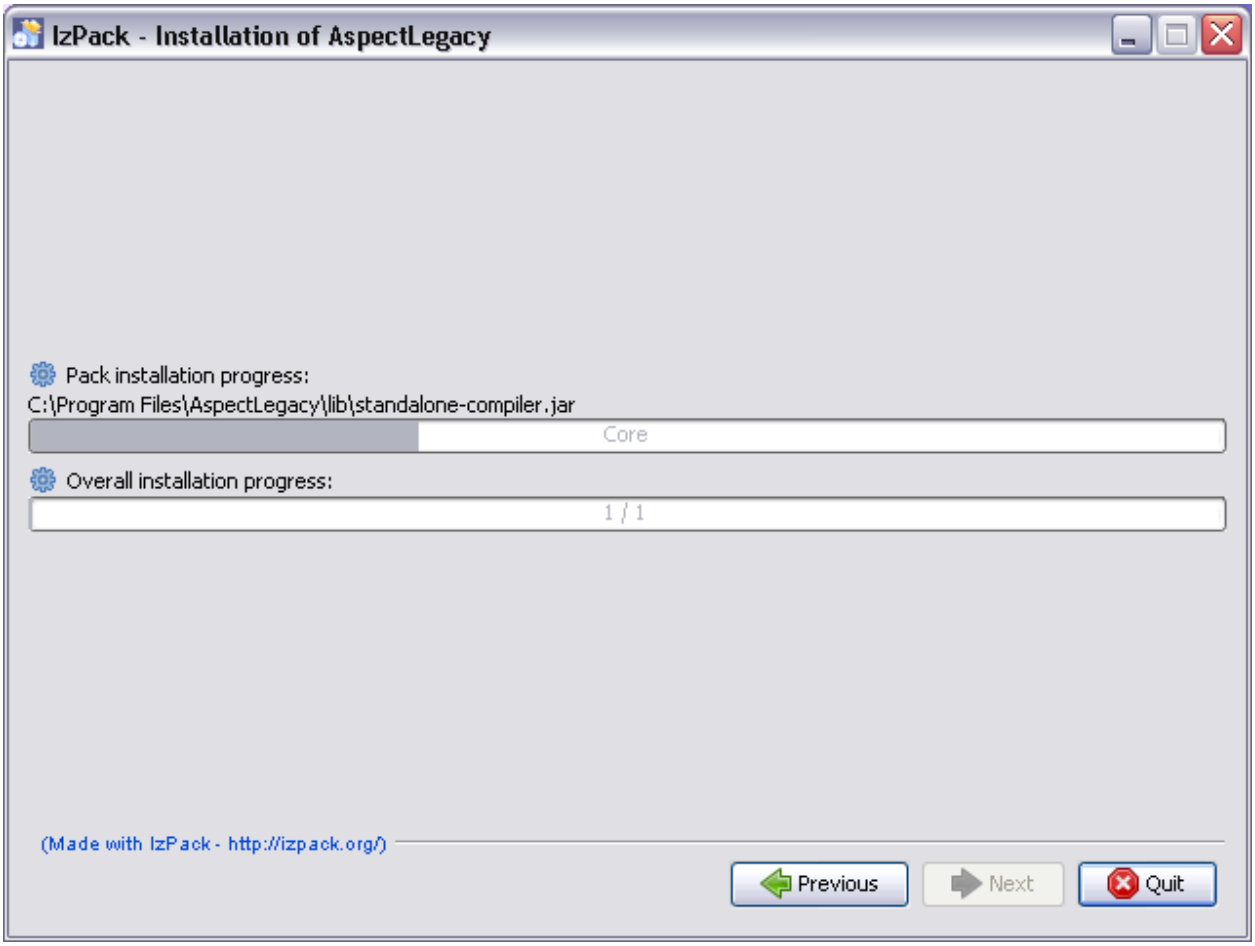
- Afterwards, you will be asked for the target location of the application files:



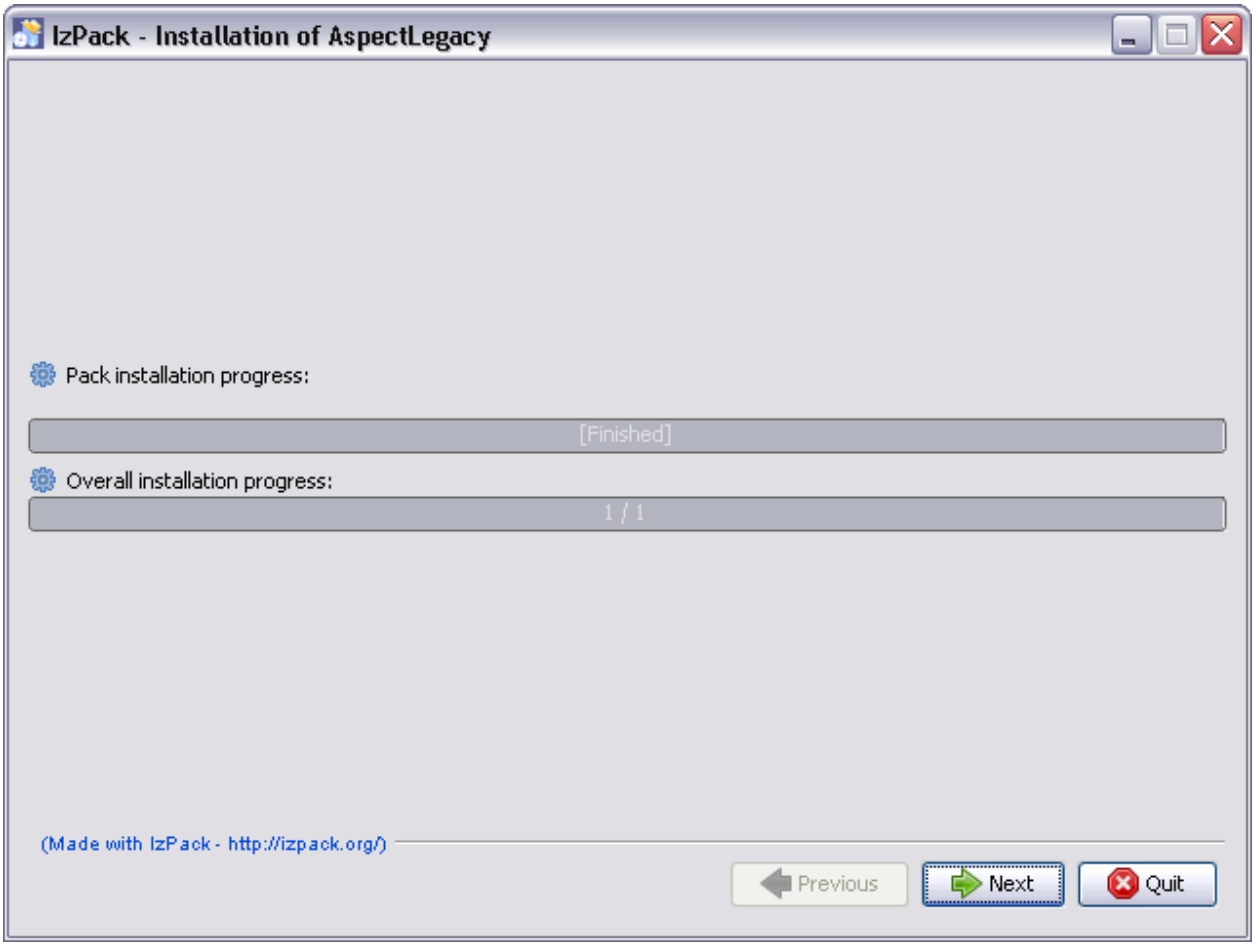
The default directory is "C:Program FilesAspectLegacy", which should be preferably chosen. In case the target directory does not exist, its creation has to be confirmed:



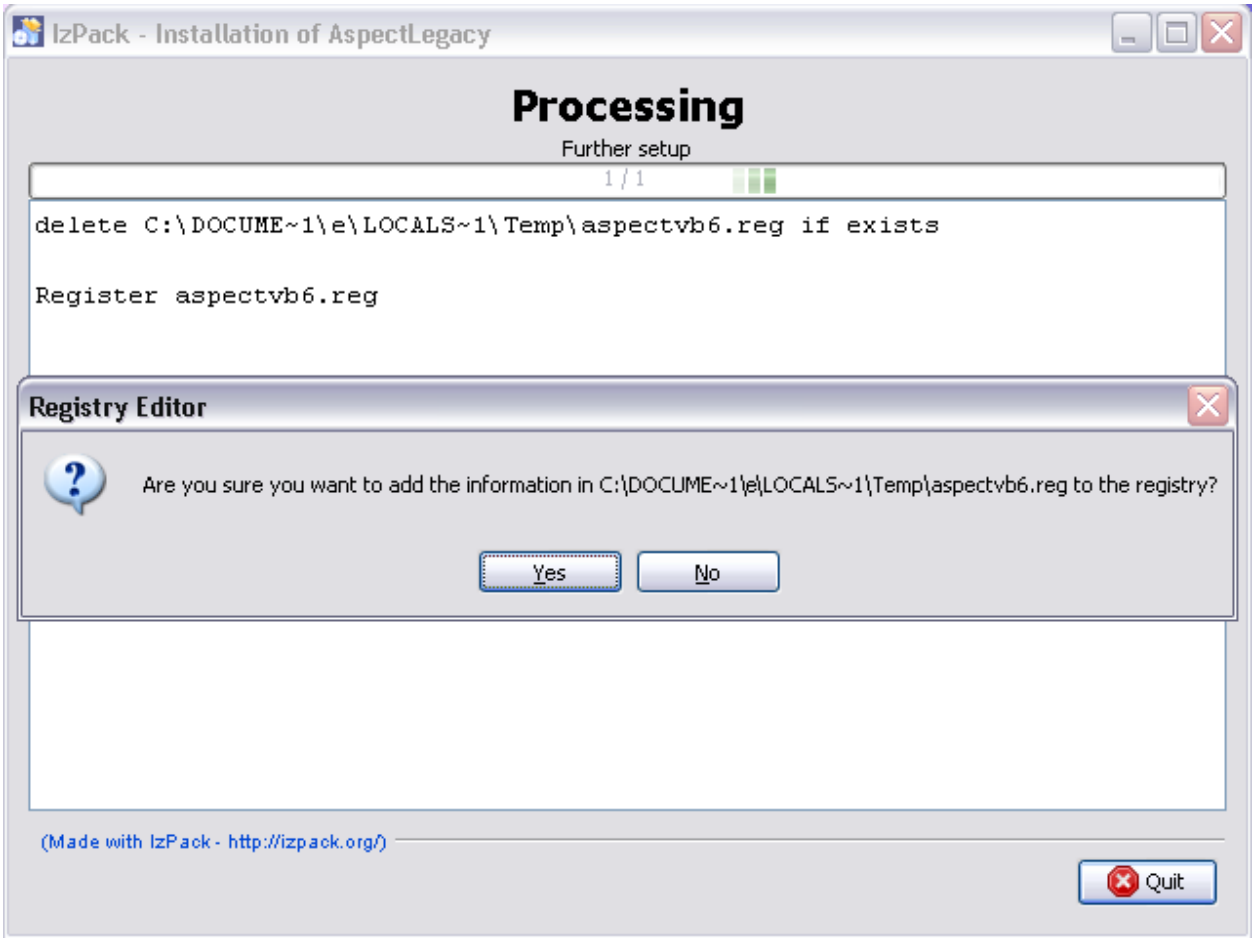
- After the target directory has been confirmed, the installation process will start:



After the installation process finishes successfully, the “Next”-button will be enabled:

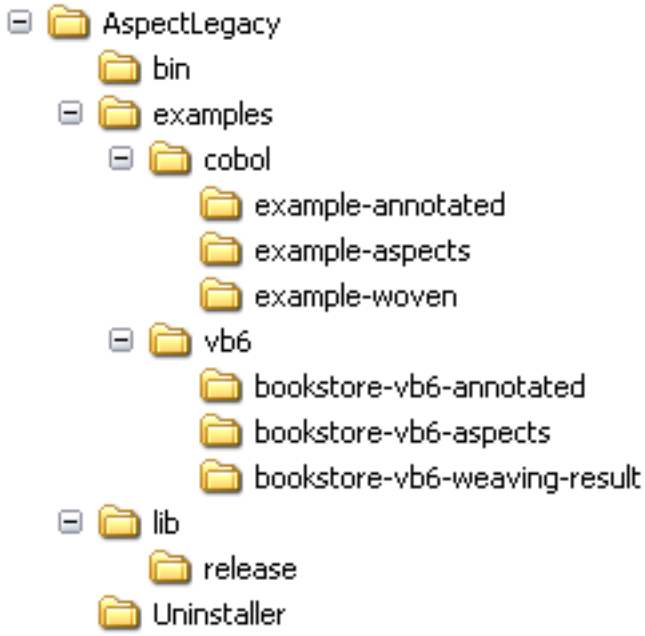


- Click the “Next” button to start the registration process. In this step, a key will be added to the registry, hence you are asked to confirm the modification:



Click the "Yes"-button for allowing the installer to add the necessary key to the registry.

- Finally, reboot your system to make the changes take effect (you might click the "Quit"-Button of the installer optionally before).
- After your system has been rebooted, the "program files" folder contains a new sub-directory with the following structure, looking similar to the one of the distribution archive:



- In the "bin" directory, you will find several batch-files, respectively the script files for starting the weaver application.

Weaving the "Bookstore" Example

It is assumed that you have already performed the steps described in the Installation section, so that an installed version of the AspectLegacy exists.

The following steps are required for starting the AspectLegacy application:

1. Go to the "bin" directory which contains several batch-files (.bat), respectively those for starting the application. You might go there by using a command-line shell or a file manager, for example "Windows Explorer".

1. Do not execute the "avb6c.bat" file directly, since this is only the core file of all other batch-files contained in the "bin" directory; instead, you might execute one of the following batch-files:

- The command

  avb6c-cmdl.bat

simply starts the application in a command-line shell. It prints a usage-overview of the available parameters to standard output, if no additional parameters are being passed.

- The command

  avb6c-cmdl.bat -g

starts the application with its GUI. The GUI will be initialized with default properties. The input-projects and the output-directory have to be defined using the file dialogues of the GUI, and the weaving process can be started by clicking the "Start"-button of the GUI (note that the input projects and an output directory have to be defined therefore). The command

avb6c-gui.bat

has been added for convenience reasons and is equivalent to this.

Now, we are going to weave the Visual Basic 6 projects of the "Bookstore" example:

1. Go to the “examples” directory which contains the batch-files “avb6c-cmdl-example.bat” and “avb6c-gui-example”. Again, you might go there by using a command-line shell or a file manager. The sub-folders

   examplesvb6bookstore-vb6-annotated\

and

examplesvb6bookstore-vb6-aspects\

contain the annotated and aspects-projects which we are going to weave by example. Furthermore, the sub-folder

examplesvb6bookstore-vb6-weaving-result\

contains the result project as it is expected to look like after weaving.

1. Execute one of the available “.bat”-files:

- The command

  avb6c-cmdl-example.bat

starts the weaving process immediately (without GUI) for the example projects and writes the weaving result to the “Temp” folder of the local user directory. Any log-messages will be displayed as command-line output. The output files will be located in the sub-folder

Tempbookstore-vb6-weaving-result\

of your user directory. Consequently, the full path of the result project will be (in Windows) similar to

C:UsersyournameAppDataLocalTempbookstore-vb6-weaving-resultBookstore.vbp

- The command

  avb6c-gui-example.bat

starts the GUI with the example projects as default input projects and a default output directory. The weaving process can be started immediately by clicking the “Start”-button of the GUI, and any log-messages will be displayed in the GUI-specific logging window.

Enhanced parameters

This section contains an overview of additional parameters to be optionally used.

- The command

  avb6c-cmdl.bat -g gui.properties

starts the GUI of the weaver with the (optionally) given configuration properties; this is for future purposes only.

- The command

  avb6c-cmdl.bat -g gui.properties -w weaver.properties -l logging.properties

starts the GUI of the weaver as well as the command above, with individual properties for weaving and logging; these properties might be even passed, if no GUI is used.

AspectLegacy (Visual Basic 6) User Guide

Created by Andre van Hoorn, last modified on Oct 03, 2016

Usage

Configuration management

For simplifying the (re-)configuration of the weaver, its properties are stored in a certain .properties-file for each component. The following components are supported:

- The graphical user interface (GUI),
- The logging unit,
- The weaver itself.

The properties of each component are hierarchically composed; whenever the value of any property is requested, the properties-sources will be searched in the following order:

1. User-defined properties file at a file system location (passed as a parameter to the weaver, see section Command-line or GUI-based); the file might contain any re-definitions of the available properties of sources 2.) or 3.).
2. User-defined properties file (with a predefined name), located in the working directory of the weaver; the file might contain any re-definitions of the available default properties of source 3.). The required names of the properties files to be located in the working directory are as follows:

dynamod.aspectlegacy.gui.properties	(default) GUI properties, for futural usage only.
dynamod.aspectlegacy.logging.properties	(default) logging properties for selection of the messages to be logged.
dynamod.aspectlegacy.weaver.properties	(default) weaver properties for specification of general flags, selections etc.

   Each of those files contains a set of key/value pairs, as they are used in Java resource bundles. A complete overview of the available properties is given by the files themselves, since all properties have been commented completely there (just have a look).
3. Default properties always present in the weaver.

Command-line or GUI-based

The weaver can be used as a command-line tool, or it can be started with a graphical user interface (GUI) alternatively.

Starting the weaver from command-line is recommended, if only default configuration properties shall be used. The weaving process itself will be much faster, as there is no synchronization with the GUI - particularly with the log-display - necessary. On the other hand, the specification of input values to be passed as command-line parameters, e.g. definition of input projects as well as the output directory, might be prone to typos.

The GUI is useful for adjusting the default configuration to certain cases. It provides an output window for log-messages and offers a much more comfortable way of weaver configuration, for example, searching for input/output paths via file dialogues or setting flags easily by clicking their corresponding checkboxes. Furthermore, the GUI provides an additional cleanup button for removing all lately generated output files.

In both cases, the execution of the weaver requires the declaration of certain parameters. Some of them are necessary, and some of them are optional. The following parameters are available (denoted in short, long format):

-s,–source-project	Path to the source project to be woven (required, if GUI is disabled).
-a,–aspects-project	Path to the aspects project (required, if GUI is disabled).
-o,–output-dir	Path to the output directory for the woven project (required, if GUI is disabled).
-g,–gui	(Optional) usage of the graphical user interface (GUI); as an optional parameter, a properties-file might be passed for individual GUI-configuration.
-w,–weaver	(Optional) properties file for weaver configuration.
-l,–logging	(Optional) properties file for logging configuration.

The paths can either be relative to the current weaver location or absolute. The paths of the input projects might denote directories or project-files, depending on the selected language.

Options

Line-break type

The line-break type determines the newline-format of the woven source code. Since different operating systems have different character codes that represent a newline, is might be useful (and in many cases necessary) to select the newline format of the weaver output code explicitly. Four types are available:

- Windows (“rn”)
- Unix (“n”)
- Mac (“r”)
- Current OS

File encoding

The encoding-option provides the selection of the encoding which is used for the source codes of a project to be woven. This might be necessary for compiling the woven sources afterwards, since some compilers demand a certain ISO-encoding of source code. For example, The Visual Basic 6 IDE gives error messages, in case the code to be compiled is not ISO8859-1-encoded (Windows); if the weaver is started under Linux (UTF-8), you will need to choose explicitly the correct encoding in that case.

Exclude patterns for files (filename filter)

If the source- and aspects-projects both contain any files of same names (in the same relative sub-directory), a file conflict occurs. In those cases, the weaving process will stop with a conflict message, as the weaver does not know which file to take for the result output.

The filename filter supports exclude patterns for files of certain names to avoid any of those conflicts. The patterns must be passed as a list to the filter. Furthermore, the filter supports simple wildcards to exclude all files with matching names from the weaving process.

The following Wildcards are available:

“?” indicates an arbitrary, single character.

“*” indicates a sequence of arbitrary characters.

Example: The line

“*.scc”; “textfile.txt”; “Image??.JPG”; “./parent/child”

indicates the exclusion of all files ending with “.scc” from the weaving process, as well as files of name “textfile.txt” and files named “Image??.JPG”, with arbitrary characters at the question mark positions. Furthermore, the file “child” contained in the sub-directory “parent” will be excluded.

Wildcards are allowed in single file-/directory-names, but not in full file-paths (yet?). The wildcards can be combined in an arbitrary way, for example:

image?*.jpg

sorts out all .jpg-images with at least one character behind the “image” token in their name, followed by an optional sequence of further characters.

Verification options

The following verification tests might be optionally done within the weaving process:

- Project directories are not allowed to be the same:

If enabled, the weaver will test on start whether the project directories do not denote the same file system location. This test should be always enabled, since the source- and aspects-projects generally have to be located in different directories.

- Files must be located in their base directories:

If enabled, the weaver will stop its work whenever a file to be accessed is located neither in the source-project directory nor in the directory of the aspects-project. This option is for futural purposes only.

- Files of the same name are not allowed to be in both directories:

If enabled, the weaver ensures that the files to be woven or copied from the source- and aspects-projects differ from each other regarding their name. This test will be done for each file if and only if the file has not been filtered out (see section Exclude patterns for files (filename filter)).

- References have to be valid:

If enabled, the weaver will finally test whether all files of the aspects-project, referenced by any annotations, have been successfully copied to output directory.

Additional language-dependent tests might be necessary, for example, the requirement of Visual Basic 6 project-files ending with “.vbp”; those tests have to be done in the upper, language-dependent layer (see section Layered architecture of the Developer Guide).

Weaver options

The following weaver options are available:

- Overwrite output files:

If enabled, existing files located in the output directory will be overwritten with files of same name. If disabled, the weaver will stop its work in case a conflict with an existing file occurs.

- Add info-marks:

If enabled, informational comments will be inserted above each transformed code-block.

- Copy all directories:

If enabled, (possibly existing) empty directories will be copied from the source projects to the output directory. If disabled, empty directories will be ignored.

- Accept hidden files:

If enabled, hidden files will be included to the weaving process.

- Clean-up on error:

If enabled, all created files will be deleted immediately after weaving has failed. If disabled, the created (but possibly incomplete or corrupt) files will be left in the output directory.

- Compile after weaving:

If enabled, the result source codes written to the output directory will be compiled immediately after weaving; external compilers or IDEs might be required for this step.

Logging options

The following logging options are available:

- Log compiler messages:

Enables the logging of informations submitted by the compiler unit, whenever the resulting source code is going to be compiled.

- Log weaver file-access messages:

Enables the logging of informations submitted by the weaver, whenever it is going to copy or modify a file.

- Log verification messages:

Enables the logging of informations submitted by the verification unit, whenever a verification test is going to be done.

- Log code-transformation messages:

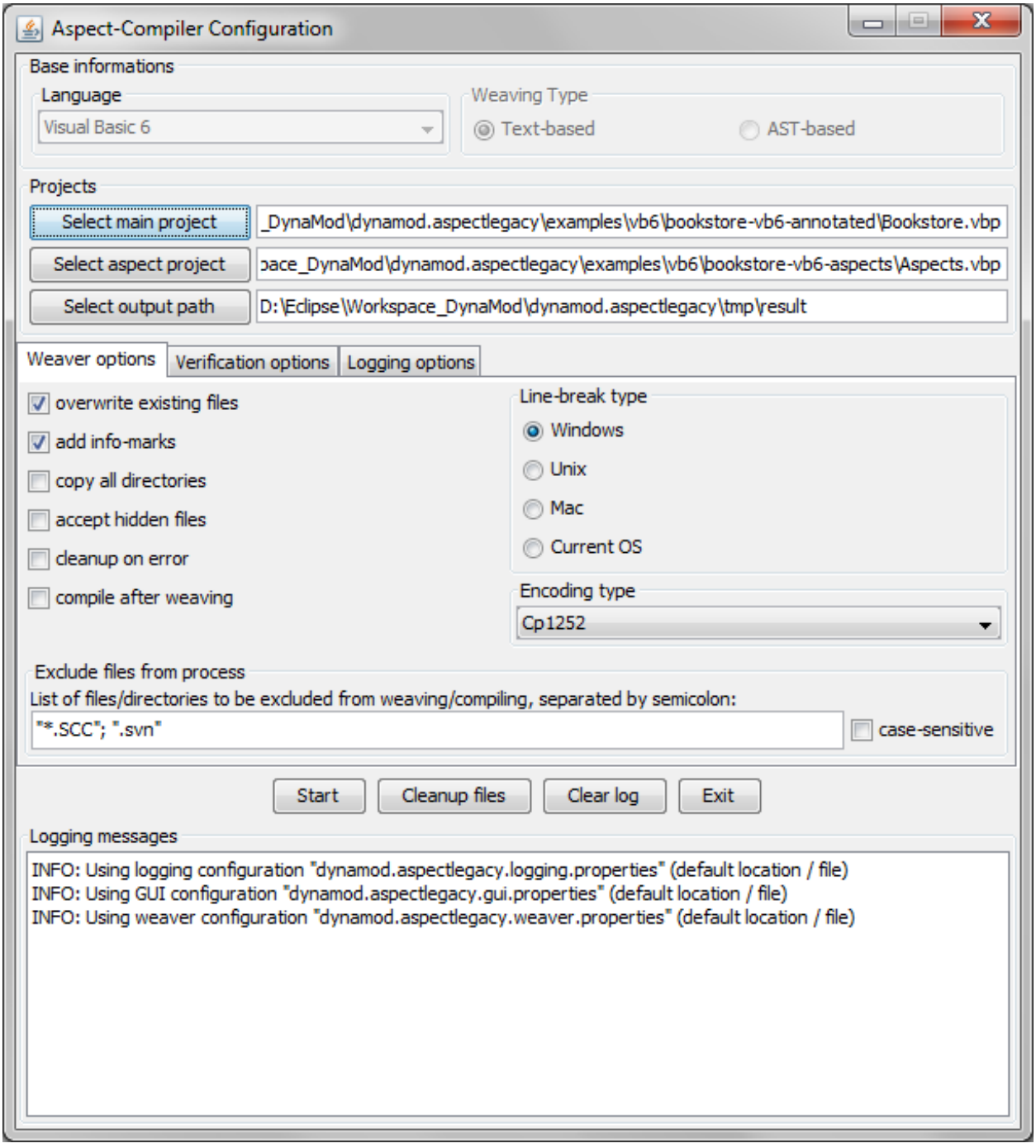
Enables the logging of informations submitted by the code-transformation unit, whenever an annotation is going to be transformed, variables are going to be inserted etc.

- Log clean-up messages:

Enables the logging of informations submitted by the clean-up unit, whenever a file or directory is going to be deleted, or even if one cannot be deleted.

The Graphical User Interface (GUI)

When the weaver application is started with the "-g" parameter (see the Quickstart for Visual Basic 6), the following configuration window will be displayed:



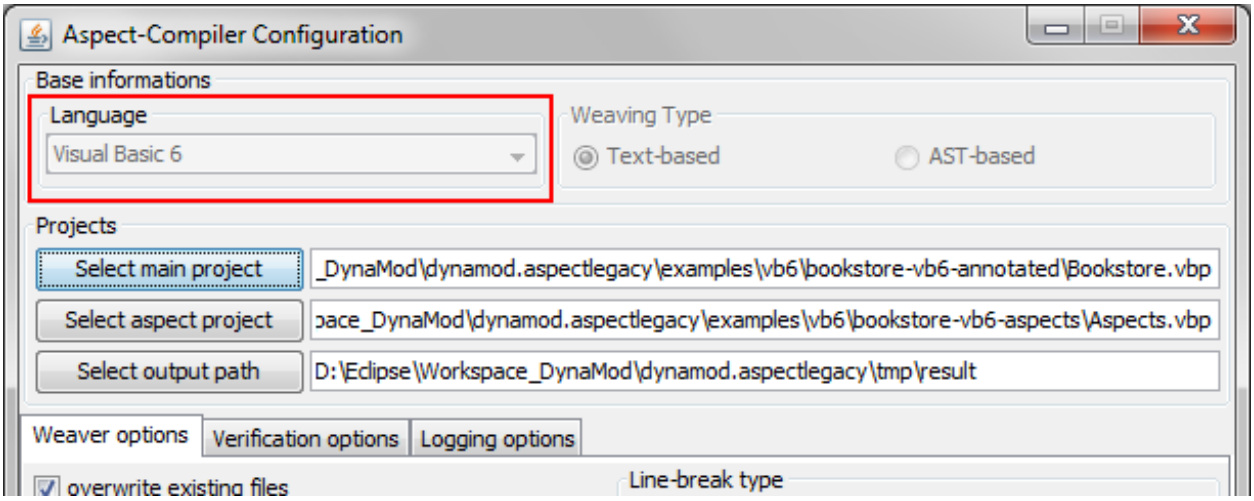
The initial settings will be in accordance with the configuration properties, as they are defined by default or transmitted by the user (see section Configuration management).

The GUI has a "top-down" design. That is, the base settings (considered language, text- or AST-based weaving type, location of project files in the file system) have to be configured in the upper part of the GUI, before the possibly language- and weaving type-specific weaver-, verification- and logging-options should be set, as well as further options in the middle part. Below the options part, a control panel contains buttons for starting and stopping the weaving process, as well as cleaning up files or exiting the application. Finally, the bottom part of the GUI contains a log-window for showing all information generated by the weaver.

The base settings need to be initialized with values for the following:

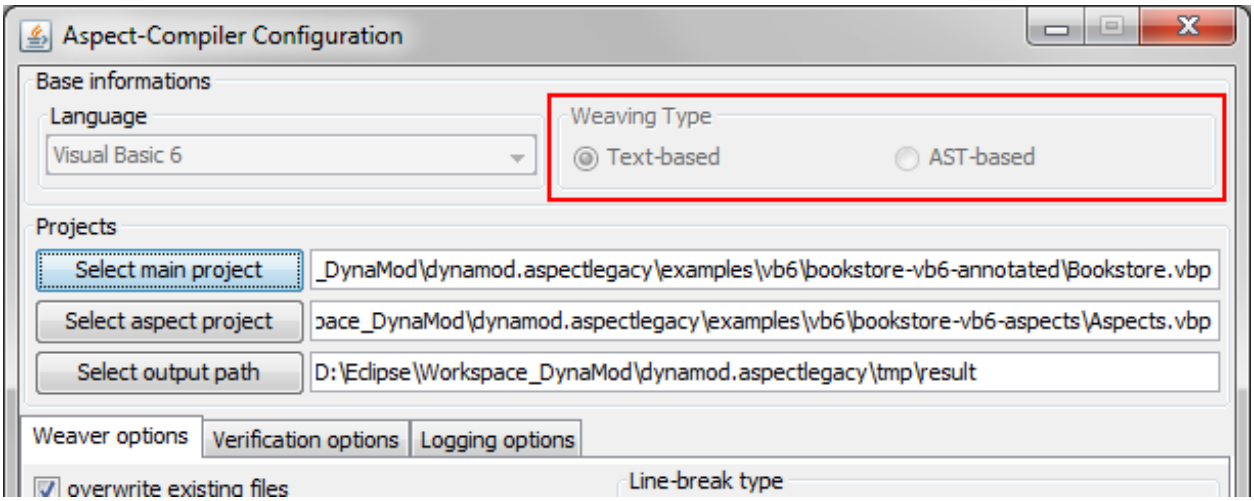
- Language (to be considered for weaving, e.g. Visual Basic 6, COBOL, . . .)
- Weaving type (text- or AST-based)
- Projects (respectively their locations in the file system)

For setting the language, a combo-box is provided, which contains all languages supported by the weaver:

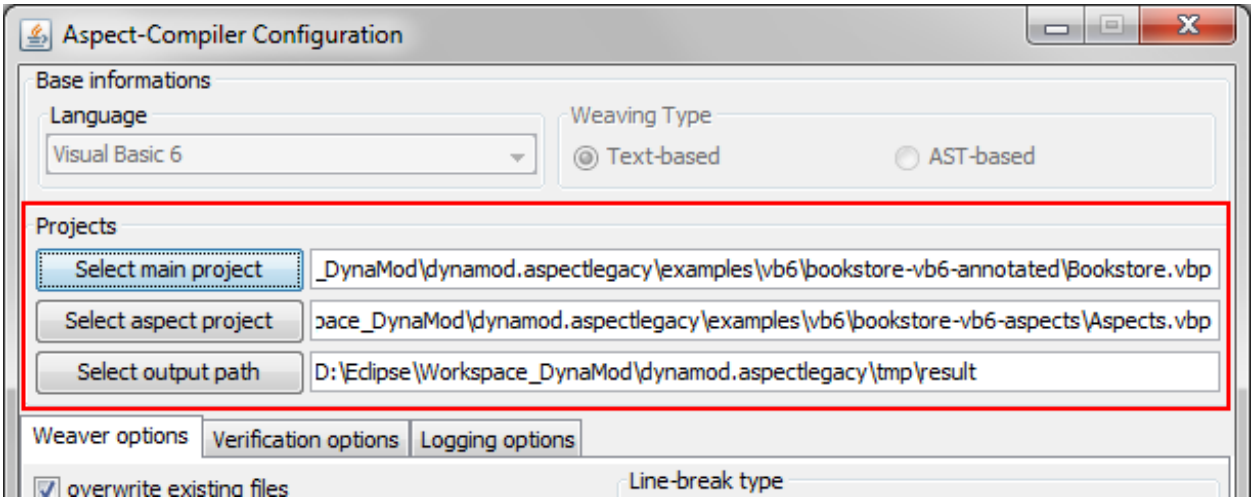


There must be at least one supported language available; if no further languages are supported, the combo-box is disabled, and the only supported language will be selected automatically.

The weaving-type must be selected by clicking the related radio button; currently, only text-based weaving is available, so this is for futural usage only:



For defining the locations of the projects within the file system, the GUI provides file-choosers, which will be shown whenever one of the “Select”-buttons is being clicked:

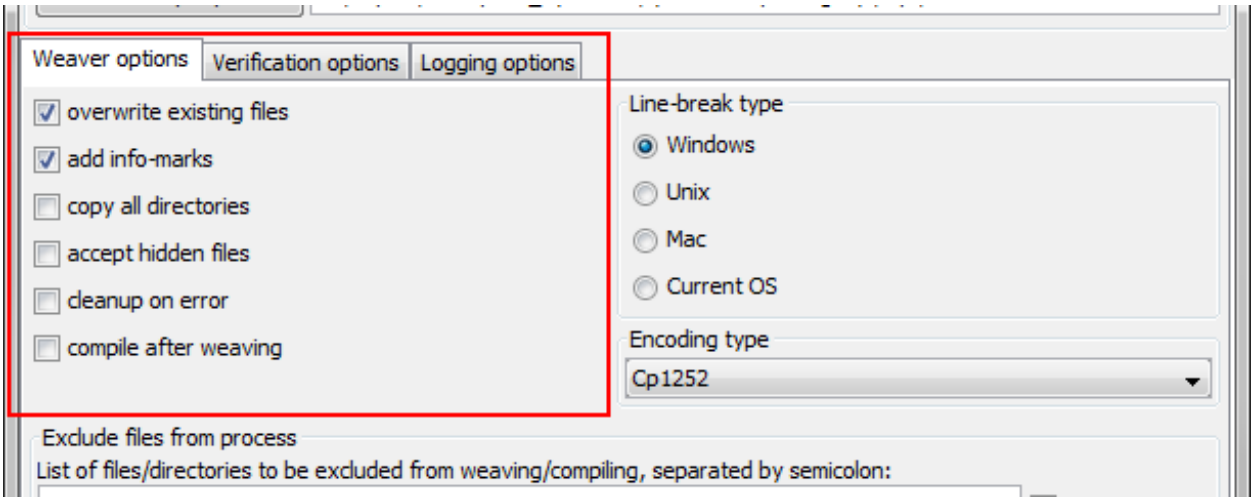


Some language like Visual Basic 6 need project-files, other languages do not. It depends on the selected language whether a project-file or a project directory must be determined.

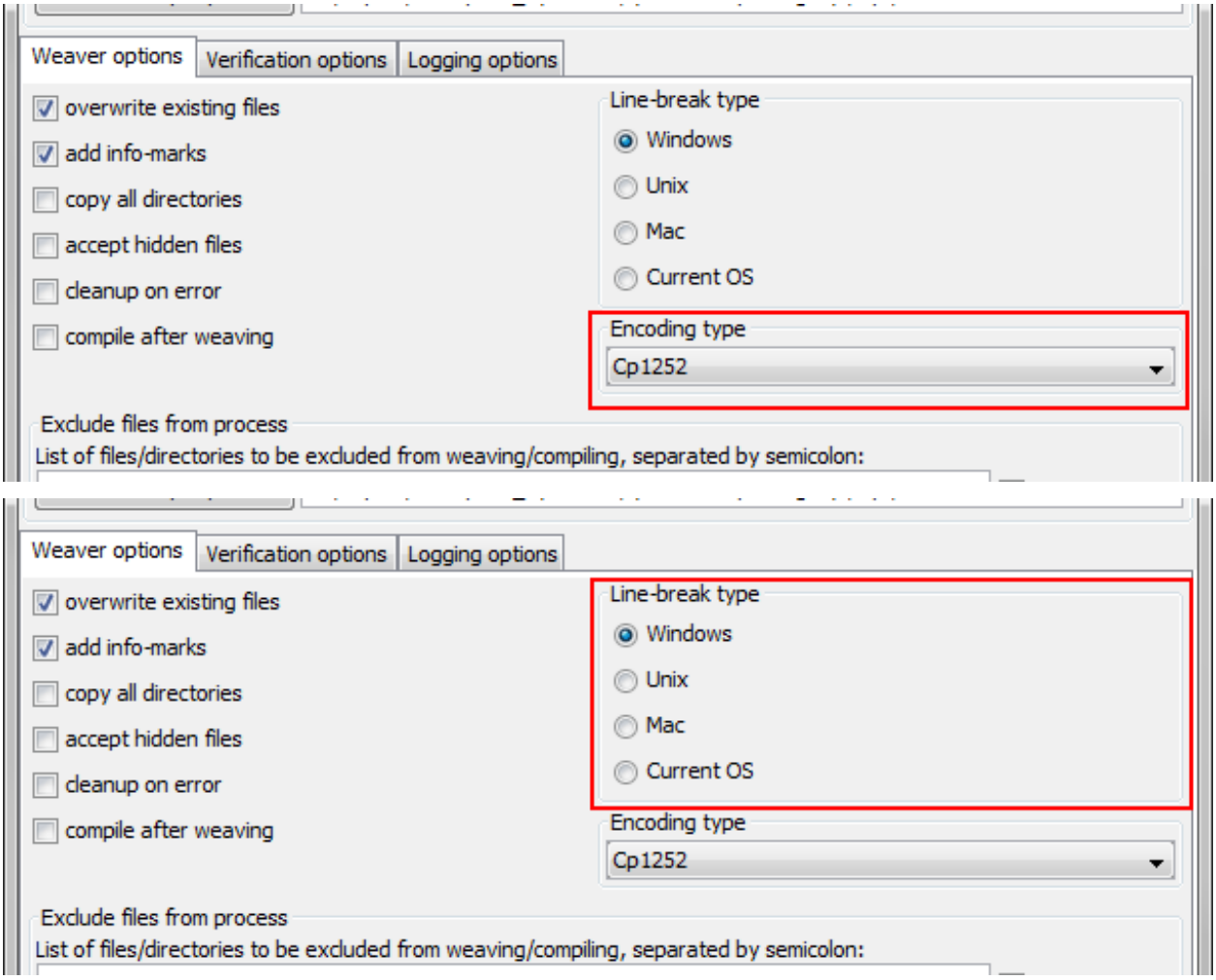
The options need to be initialized with values for the following:

- Weaver options
- Verification Options
- Logging options

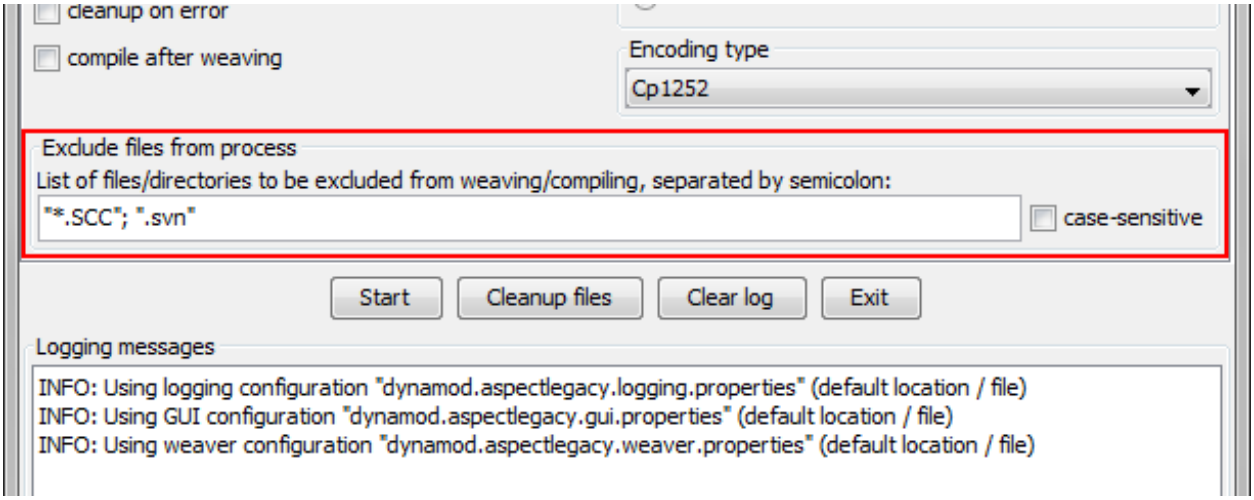
Since all of these option values are boolean, the related part of the GUI contains a tabbed overview, with a tab for each option group, supporting check-boxes for setting the values easily:



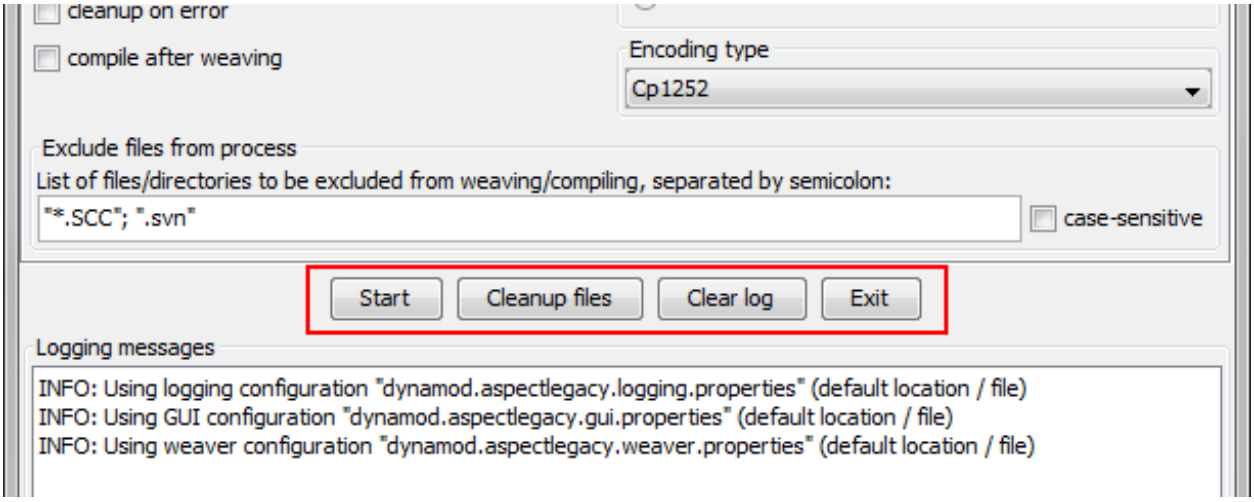
Furthermore, the options panel provides input masks for the encoding type as well as for the line-break type to be used:



Finally, an input field for exclude file patterns is included (see section Exclude patterns for files (filename filter)):



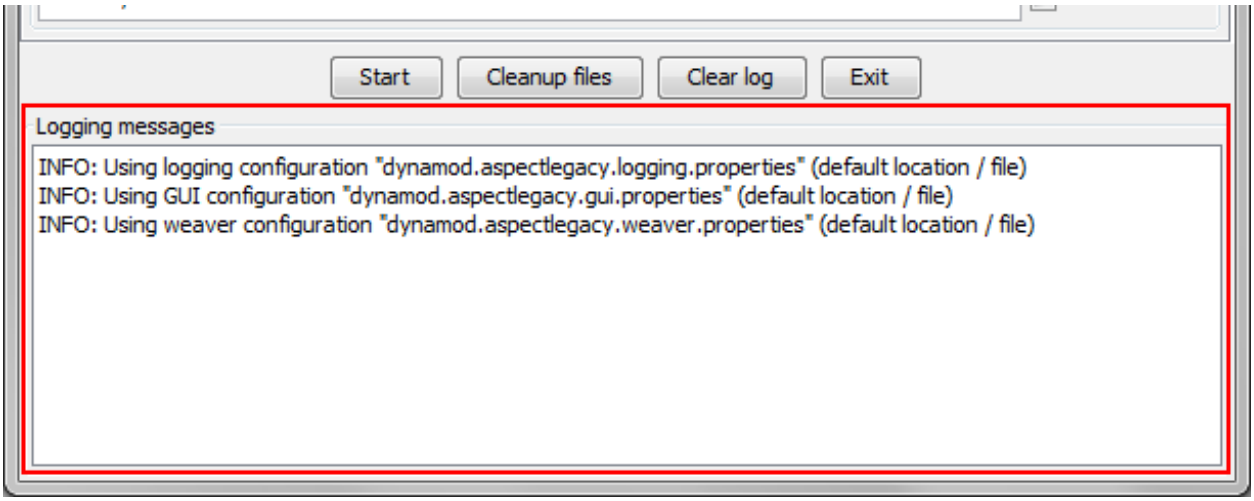
The control panel holds the control for all processes to be started or stopped:



The following options are available:

Start	• Start the weaving process.
Cleanup files	• Delete all newly created files/directories of the latest weaving-process; this will not delete any files/directories, which existed already before.
Clear log	• Clear the logging display (see section Logging window).
Exit	• Quit application.

The logging window displays the information generated by any running task of the weaver:



The content will be displayed multi-coloured, whereas the colours are assigned as follows:

Black	• General informations (e.g. confirmation message for a finished process).
Light red	• Fatal errors (whenever an exception makes the weaver stopping an operation).

Optional log-messages (see section Logging options):

Dark red	• Clean-up messages.
Green	• Compiler messages.
Purple	• Verification messages.
Dark blue	• Code-transformation messages.
Light blue	• Weaver file-access messages.

Limitations / Future Work

As a futural feature, weaving might be done text-based or AST-based (see section Weaving type)). The following section describes the main differences between both types.

Text-based weaving

In the text-based weaving mode, the weaver will scan the source code line-by-line and generate the transformed code "on the fly", without syntax parsing. Syntax analysis is restricted to single lines, as they are read while weaving.

AST-based weaving (futural feature)

In case AST-based weaving is selected, the source code of the input projects will be parsed for generation of an abstract syntax tree. This enables the detection of multiple-rows-comments and consequently the detection of line-breaks, GOTOs etc.

Text-based weaving vs. AST-based weaving

Text-based weaving is strongly restricted, since the source code is considered only as plain text. There is no extensive analysis of syntax/semantics, hence even the partial analysis of the source code is difficult or not possible. For example, in languages like C/C++ or Java, where comments might be nested or be wrapped over several lines, no certain conclusion can be drawn about a single text line (e.g. whether a certain line belongs to a wrapping comment, even if the statement itself seems to be a command).

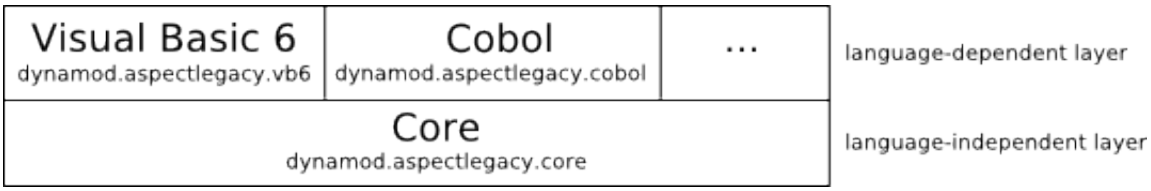
Consequently, text-based weaving should be primarily used for "simple" languages, particularly for those which allow exclusively single-line-comments (e.g. COBOL).

AST-based weaving should be used to ensure that the source code is parsed correctly, for the ability to detect "complex" syntactical constructs split on multiple lines (comments, split commands etc.).

AspectLegacy (Visual Basic 6) Developer Guide

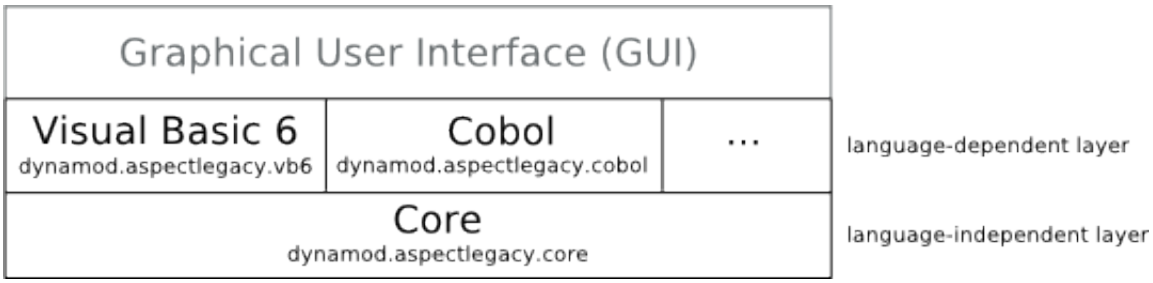
Layered Architecture

For adapting the weaver framework to any additional programming language, its architecture consists of a generic core component, which serves as a base unit for any language-specific weaver adaption. Consequently, the architecture of the weaver algorithm can be illustrated as a two-layers model, consisting of a constitutive, language-independent layer (core) and an adapting language-dependent layer for any related programming language:



The lower layer (which can be considered a single core-component, since it is implemented as an own Java package) provides the generic functionality, e.g. file access, search for annotations within code, verification and logging. The upper, language-dependent layer provides the language-dependent functionality.

Additionally, an optional Graphical User Interface (GUI) is provided for simplifying the weaver configuration process. The GUI can be seen as a third layer, covering all language-dependent implementations, since it provides the configuration functionality for all currently supported languages:



Components of the core layer

Since the core layer (= Java package) provides the generic functionality of the weaver, it includes particularly the search-and-insert-algorithm for text-based weaving and generic support for code compilation. Consequently, it consists of two sub-components, one for the weaving process itself and another sub-component for (optionally) compiling the result code afterwards; each of those components denotes an own Java sub-package of dynamod.aspectlegacy.core:

- Code weaving unit (package dynamod.aspectlegacy.core.weaver)
- Code compilation unit (package dynamod.aspectlegacy.core.compiler)

The weaving unit provides the generic weaving functionality, which is copying or reading files, searching annotations within source code, substituting annotations with their indicated code, writing output files, verification, logging and possible clean-up of files. It includes four sub-components, each one denoting an own Java sub-package of dynamod.aspectlegacy.core.weaver:

- Clean-up unit (package dynamod.aspectlegacy.core.weaver.cleanup)

- File access unit (package dynamod.aspectlegacy.core.weaver.fileaccess)
- Code transformation unit (package dynamod.aspectlegacy.core.weaver.transformation)
- Verification unit (package dynamod.aspectlegacy.core.weaver.verification)

The tasks within the weaving process are assigned to the units as follows:

- The *clean-up* functionality includes the (optional) deletion of any files and directories which have been created while weaving.
- The *file access* unit provides all input-/output-operations, where files might even denote directories. File access includes the reading of directory content, copying files, reading (text-)file content etc.
- *Code transformation* is the search for annotations in a given source code and substitution of those with the code they indicate.
- *Verification* is done for checking possible violations of any contrains. This includes, for example, the requirement of having different directories for the source- and aspects-projects, or even the existence of certain output files.

The package for code compilation contains a facade class, which summarizes all compiler options. Since the compiler unit makes use of the weaver unit, it additionally delegates certain method calls to the facade provided by the weaver package.

https://build.se.informatik.uni-kiel.de/DynaMod-tools/trac/attachment/wiki/dynamod.aspectlegacy/DeveloperGuide/weaver_core_packages.png

Code compilation will be usually done by invoking an external compiler or IDE. Therefore, the configuration files of the weaver might be adjusted (see section Configuration management of the User Guide).

Language-dependent enhancements through the upper layer

Adapting the core package to any certain programming language requires some work, but the aim of this framework is to keep the implementation effort restricted to just a few classes.

Several core components contain interfaces and abstract classes with abstract methods to be implemented. Most of these methods provide simple functionality, for example, detection of comment-indicators and removing them from code lines; the latter being necessary for dealing with annotations as single-line-comments. The abstract methods have to be implemented by the upper-layer classes of the weaver model, since those classes provide the language-dependent functionality of the weaver. Additionally, certain core-interfaces define the methods which will be invoked by the code transformation unit, whenever an annotation is to be transformed. These interfaces must be implemented, too, since the code constructs to be inserted in place of the annotations depend on the considered language. The implementation of the transformation part takes some effort, but afterwards the weaver is nearly complete. Finally, the set of files to be included into the weaving process must be determined.

For each programming language to which the core package shall be adapted, the following steps must be done:

1. The (abstract) class AbstractCodeLine.java represents a single, generic code line. It contains abstract methods for checking whether a given String is a comment, for removing a comment-indicator from a given comment-String and for cloning a code line itself. Note that only single-line-comments are supported in text-based weaving, which includes for example lines with leading "//" in Java, "REM" or "'" in Visual Basic 6, or "*" in Cobol (see section Limitations/Future Work of the User Guide). For making this class language-dependent, the methods mentioned above must be implemented by a sub-class, related to the considered language. A good way for implementing the required functionality of these classes is the use of regular expressions.
2. Interface ICodeLineFactory.java serves, as its name indicates, as a factory for code lines. It provides methods for creating instances of concrete AbstractCodeLine-subclasses, implemented in step 1. The implementation of the factory-methods is mostly trivial (just return new instances of code lines).

3. Interface IAnnotationTransformer.java is the base interface for any code transformation indicated by an annotation. Each annotation type (e.g. "CALL", "EXECUTION") requires a type-related annotation transformer to be implemented, since each type indicates a different kind of code transformation. Any implementation of this interface needs some more comprehensive effort, as the code transformation includes the substitution of annotations with their indicated, language-dependent code.

   The interface contains the transform()-method, which will be invoked by the code transformer, whenever it finds a new annotation to be transformed. Hence, the transformer passes amongst other parameter values the annotation itself, the index of the first line to be transformed within the original code (which is usually the line just after the related annotation). Furthermore, it provides the original code as well as the transformed code (as it is in the current state). The remaining parameters of this function are only relevant for recursive function-calls.

   The transform()-method must analyse the given annotation and add the indicated code to the end of the currently transformed code (note that the text-based transformation is done top-down, so additional transformed code will just be appended to the currently existing list of transformed lines); the original code must be left unmodified. Finally, the function has to return the index of the next line within the original code content to be examined.

4. While weaving, a certain set of files must be read and written. Therefore, the package dynamod.aspectlegacy.core.weaver contains the (abstract) class AbstractFileCollector.java, which defines four abstract functions to be implemented for defining the considered set of files:

   - getWeavableMainProjectFiles()
   - getMainProjectFilesToBeCopied()
   - getWeavableAspectProjectFiles()
   - getAspectProjectFilesToBeCopied()

   The getWeavableFiles()-methods must return the lists of source-files contained in the main/aspects projects, and the getProjectFilesToBeCopied()-methods have to return the "non-weavable" files, like images, audio-files etc.; the sets of files returned by these methods must be disjunct, and their union must include all required files for generating the output project.

   The class FileCollectorAdapter.java of the package dynamod.aspectlegacy.core.weaver provides methods for collecting the required files by their file endings from certain directories.

Besides the transformation of annotations, additional (language-dependent) transformation might be necessary, depending on the chosen language (e.g. insertion of new, global variables). Therefore, the visibility of certain methods in class CodeTransformer.java of the core.transformation package is "protected", so that these methods can be accessed by any sub-class (see comments within the source code).

Once you have done the steps above, the abstract classes

- dynamod.aspectlegacy.core.weaver.AbstractAspectWeaverCreator.java and
- dynamod.aspectlegacy.core.compiler.AbstractCompiler.java

have to be implemented. The implementation of the abstract methods within these classes is mostly trivial (just return new instances of the classes you have implemented by doing the steps above). Additional functionality might require the overwriting of certain methods within these classes, but this is case-dependent. For example, this includes generating project-files (".vbp") for Visual Basic 6 projects as it is done in the example source code.

Note: This is legacy documentation. There might be discrepancies between the documentation and the current version of the external software used when developing this **Kieker** extension.

1.4.7 Kieker4NET

Kieker4NET is a Kieker adapter supporting monitoring of programming languages based on Microsoft's .NET platform. The adapter has been developed as a part of the DynaMod research project. It has been tested particularly with C#.

..note:

```
This adapter has not been used for some time.
```

Installation of Kieker4NET

Requirements

1. ' JNBridgePro <http://www.jnbridge.com/>'__ license
2. PostSharp license

JNBridge Download and Licensing

JNBridePro can be downloaded from ' http://www.jnbridge.com/bin/downloads.php?pr=1&id=0 <http://www.jnbridge.com/bin/downloads.php?pr=1&id=0>'__. You'll receive an evaluation version with a trial license. which will remain functional for 30 days. After having submitted a registration form, JNBridgePro is available for 32-bit and 64-bit (untested with Kieker4NET) versions.

The following JNBridePro license types are available:

1. *Developer license:* Required for developing Kieker4NET.
2. *Deployment license:* Required for distributing/installing Kieker4NET. Note, that deployment license are only available if a developer license has been purchased before. Please note that non-OEM (default) and OEM licenses are available.

For details on the JNBridgePro licensing, see ' http://www.jnbridge.com/store.htm <http://www.jnbridge.com/store.htm>'__.

Installing the JNBridge License

Having registered via the ' JNBridge download page <http://www.jnbridge.com/bin/downloads.php?pr=1&id=0>'__, you should receive an e-mail with more information on the product, including the *activation key*.

Install JNBridgePro by running the setup wizard provided by the downloaded JNBSetup6_0_x86.msi file. During the setup wizard, you'll have to select one of the following configurations:

1. *Development configuration*: deployment configuration + proxy generation tool and demos.
2. *Deployment configuration*, including only the Java and .NET runtime components.

The installer installs two JNBridgePro versions, one for .NET 2.0/3.0/3.5 and a second for .NET 4.0. The development configuration is required on the machine creating the .NET proxy for the kieker-<version>.jar. It also include the JNBridePro plugins for Visual Studio 2005, 2008, and 2010.

1.4.8 Related Topics

- instrumenting-software-adaptive-monitoring
- Creating Probes
- Creating new Event Types

1.5 Analyzing Monitoring Data

Todo: References and links must be refreshed.

In this section, we discuss the use of existing tools to analyze monitoring data, a way to compose your own analysis based on existing Kieker analysis stages utilizing the TeeTime pipe and filter framework, and how to create new filters within.

- analyzing-composing-analysis-tools
- analyzing-writing-your-own-analysis-stage
- architecture-java-analysis-and-tools-api

1.6 Kieker Tools

All tools can be found in the binary bundle (`kieker-1.15-binaries.zip`) in the `tools` directory. The `tools` directory contains a set of tools prepacked as tar and zip archives. Each archive contains one tool with all its libraries and start scripts. The start scripts are located in the `bin` directory and the libraries in the `lib` directory. In the tool root directory, e.g.,`trace-analysis-1.14`, you can find a `log4j.cfg` file, used to configure the logging output for your tool. The `bin` directory contains two scripts one named after the tool usable in Linux, FreeBSD, MacOS, etc. and one with `.bat` extension for Windows.

To change the logging setup you can either change that file or define additional options with the JAVA_OPTS environment variable, e.g.,

`export JAVA_OPTS="-Dlogback.configurationFile=/full/path/to/logger/config/logback-trace.groovy"` or use the tool specific `_OPTS` variable, e.g., `TRACE_ANALYSIS_OPTS` for the `trace-analysis` tool.

Furthermore, you can use both variables to pass additional JVM parameters and options to a tool.

- kieker-tools-webgui (deprecated)
- kieker-tools-trace-analysis-tool
- kieker-tools-trace-analysis-gui (deprecated)
- kieker-tools-convert-logging-timestamps
- kieker-tools-log-replayer
- kieker-tools-collector
- kieker-tools-kdb (deprecated)
- kieker-tools-resource-monitor
- kieker-tools-irl

- kieker-tools-dot-pic-file-converter

Please note there are other tools available for Kieker which are not bundled with Kieker.

1.7 Developing with Kieker

In this section we discuss how to develop your own analyses with Kieker and embed them in tools and services. We will reference to architecture documentation and JavaDoc when needed. As this is a living software project, there might be a discrepancy between API documentation, architecture and the documentation in this section. In that case, always refer to the API.

- developing-with-kieker-writing-tools-and-services
- developing-with-kieker-writing-ui-and-web-tools

1.8 Extending Kieker

Sometimes a Kieker probe or a Kieker stage (filter) may not provide the necessary features you have in mind. In that case, you can extend Kieker. In this section we explain how to extend Kieker in many different ways:

- How to write new
  - Records
  - Probes
  - Stages
  - Features of Stages
  - Serializaion Formats
- How to support new programming languages
  - extending-kieker-general-language-and-platform-support
  - architecture-file-and-serialization-formats

1.9 Architecture

For certain parts of Kieker, we created architecture documentation to support the use of Kieker and development for Kieker.

1.9.1 Java

- **java-monitoring-api**
  - java-monitoring-controller-api
  - java-writers-api
  - java-probes-api
- **java-analysis-api**
  - java-readers-api

- java-records-api

1.9.2 Generic

- architecture-file-and-serialization-formats

1.10 Lectures

This tree contains tutorials presented at events such as guest lectures, conferences, etc.

- lectures-icpe-dublin
- lectures-university-pavia

1.11 Related Work

1.11.1 Monitoring Tools (commercial / non-research)

- AppDynamics
- Btrace
- CA Wily Introscope
- DynaTrace
- Foglight
- IBM Tivoli Monitoring
- JAMon
- Java Simon - Simple Monitoring API
- JETM
- JINSPIRED JXInsight/OpenCore
- Metrics
- MonALISA: MONitoring Agents using a Large Integrated Services Architecture
- MoSKito: Health and Performance Monitoring for Java Applications
- Munin (infrastructure/system-level monitoring; similar to like Nagios?):
- New Relic
- NovaTec inspectIT
- Nagios
- Perf4J
- Replay Solutions
- RHQ
- Software Diagnostics: Application Logger
- Software-EKG

- Vector by Netflix
- Zabbix (server + infrastructure monitoring?)

1.11.2 Monitoring Tools (research)

- COMPAS JEEM (T. Parsons, A. Mos, and J. Murphy. Non-intrusive end to end run-time path tracing for J2EE systems)
- Dyper
- Magpie (P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems)
- Rainbow (S.-W. Cheng. Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation)
- SPASS-meter (Univ. Hildesheim, Germany)
- Libmonitor

1.11.3 Performance/Monitoring Tools Web Sites

- SPEC Research Group
- http://www.monitortools.com/
- http://www.opensourcetesting.org/performance.php

1.11.4 Dynamic Reverse Engineering Tools

- Reverse Java

1.11.5 Log Analysis

- Graylog2

1.11.6 Repositories of Performance Data

- http://trust.salesforce.com/

1.11.7 Profilers

- JBoss profiler
- JFluid/NetBeans Profiler
- Criterion

1.11.8 UML Graph Libraries

- UMLGraph

1.11.9 Instrumentation Tools

- Pin (see also ATOM)
- DiSL
- FERRARI : Framework for Exhaustive Rewriting and Advanced Runtime Instrumentation

1.11.10 ARM: Application Response Measurement

- https://collaboration.opengroup.org/tech/management/arm/
- http://dx.doi.org/10.1109/IWSM.1998.668123
- OpenARM: http://open-arm.sourceforge.net/

1.11.11 Trace/Control Flow Analysis/Visualization

- Fraunhofer SAVE (Software Architecture Visualization and Evaluation)
  - "a research prototype for goal-oriented analysis of software systems. Its primary feature is architecture compliance checking" (http://www.eclipsecon.org/summiteurope2009/sessions?id=1055)
  - "SAVE supports the analysis of runtime traces of instrumented software systems in formats based on Eclipse TPTP (Test & Performance Tools Platform), AspeCt C (ACC), or Comma Separated Values (CSV)." (http://www.iese.fraunhofer.de/de/Images/SAVE_e_2009_tcm122-46390.pdf)
- HPI, Computer Graphics Systems group: http://www.hpi.uni-potsdam.de/doellner/index.html
  - Trümper, Jonas and Bohnet, Johannes and Döllner, Jürgen: Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In Proceedings of the ACM Symposium on Software Visualization, pp. 133-142, 2010. (http://www.hpi.uni-potsdam.de/doellner/publications/year/2010/1219/TBD10.html)
  - Trümper, Jonas and Bohnet, Johannes and Voigt, Stefan and Döllner, Jürgen: Visualization of Multithreaded Behavior to Facilitate Maintenance of Complex Software Systems. In Proceedings of the International Conference on the Quality of Information and Communications Technology, pp. 325-330, 2010. (http://www.hpi.uni-potsdam.de/doellner/publications/year/2010/1218/TBVD10.html)
- AppDynamics (Application Management for the Cloud Generation) (http://www.appdynamics.com/products-features-and-benefits.php)
- Dr. Garbage Tools (http://drgarbagetools.sourceforge.net/, http://dx.doi.org/10.2316/P.2012.790-033)

1.11.12 Use Cases for Dynamic Analyis

- Profiler-guided optimization
- Monitoring-oriented programming
- ...

1.11.13 Application/User-Space Monitoring in Linux

- UProbes/UTrace
- trace-cmd/libtracevents

CHAPTER 2

Licensing

Kieker is licensed under the Apache License, Version 2.0. You may obtain a copy of the license at <http://www.apache.org/licenses/LICENSE-2.0>

The **Kieker** source and binary release archives include a number of third-party libraries. Appendix~ref{appendix:libraries} lists these libraries along with information on the licenses. The `lib/` directory of the release archives contains a `.LICENSE` file for each third-party library, pointing to the respective license text.

CHAPTER 3

Citing Kieker

When referencing Kieker resources in your publications, we would be happy if you respected the following guidelines:

- When referencing the Kieker project, please cite our IPCE-2012 paper and/or our 2009 technical report [TR-0921]. Also, you might want to add a reference to our web site (<http://kieker-monitoring.net/>) like

```
@MISC{KiekerWebSite,
  author = {{Kieker Project}},
  title = {Kieker web site},
  year = CURYEAR,
  url = {http://kieker-monitoring.net/}
}
```

- When referencing this user guide, e.g., when reprinting contents, please use a proper citation.