The OW2 Consortium

Developer's Guide

Authors:

P. Garcia	(INRIA)
M. Leclerq	(ST Micro-Electronics)
V. Quéma	(CNRS)
J.-B. Stefani	(INRIA)

Released	January 2008
Status	Draft
Version	2.0

General Information

Please send comments on this document to `dream@objectweb.org`. Authors would be glad to hear from people using or extending Dream.

Trademarks

All product names mentioned herein are trademarks of their respective owners.

Requirements

Before reading this document, be sure you have following knowledge :

- The FRACTAL component model (`http://fractal.objectweb.org`).
- The Eclipse Development Environment (IDE) for Java developers (`http://www.eclipse.org`).
- The SVN version control system(`http://subversion.tigris.org`).

Contents

List of Figures

1 Configuring Eclipse and getting Dream sources

1.1 Getting Eclipse and required plugins

The latest eclipse release is available at the following URL: `http://www.eclipse.org/downloads/`.

We advice you to install the following plugins:

- Subclipse is a plug-in providing support for Subversion within the Eclipse IDE. Installation instructions are available at the following URL: `http://subclipse.tigris.org/install.html`.
- Checkstyle is a plugin that helps you ensure that your Java code adheres to a set of coding standards. Installation instructions are available at the following URL: `http://eclipse-cs.sourceforge.net/`.
- Maven plugin provides integration for Maven, an open-source build manager for java projects. Installation instructions are available at the following URL: `http://m2eclipse.codehaus.org/`

1.2 Configuring the SVN repository

- Open the "SVN perspective": `Window` > `Open Perspective` > `Others` > `SVN Repository Exploring`.
- In the "SVN Repositories" view, click on the `Add SVN Repository` button (see figure 1).
- Fill in the dialog box shown in figure 2 with :
  - `svn://svn.forge.objectweb.org/svnroot/dream` for anonymous (read-only) access.
  - `svn+ssh://developername@svn.forge.objectweb.org/svnroot/dream` for developer access.

1.3 Checking out the Dream *trunk* module

The `trunk` module contains everything you need to develop dream : DREAM modules and a `tools` module to configure eclipse. The *dream* module is an alias of the three base modules (*dreamcore*, *dreamadl*, and *dreamlib*).

N.B. : the `trunk` module is the main development branch of the DREAM project. To develop a specific module, the best practice is to develop first in a separate branch and then merge the branch to the trunk. For more information about branching under SVN, see `http://svnbook.red-bean.com/en/1.1/svn-book.html#svn-ch-4`.

To check out the *trunk* module, proceed as follows:

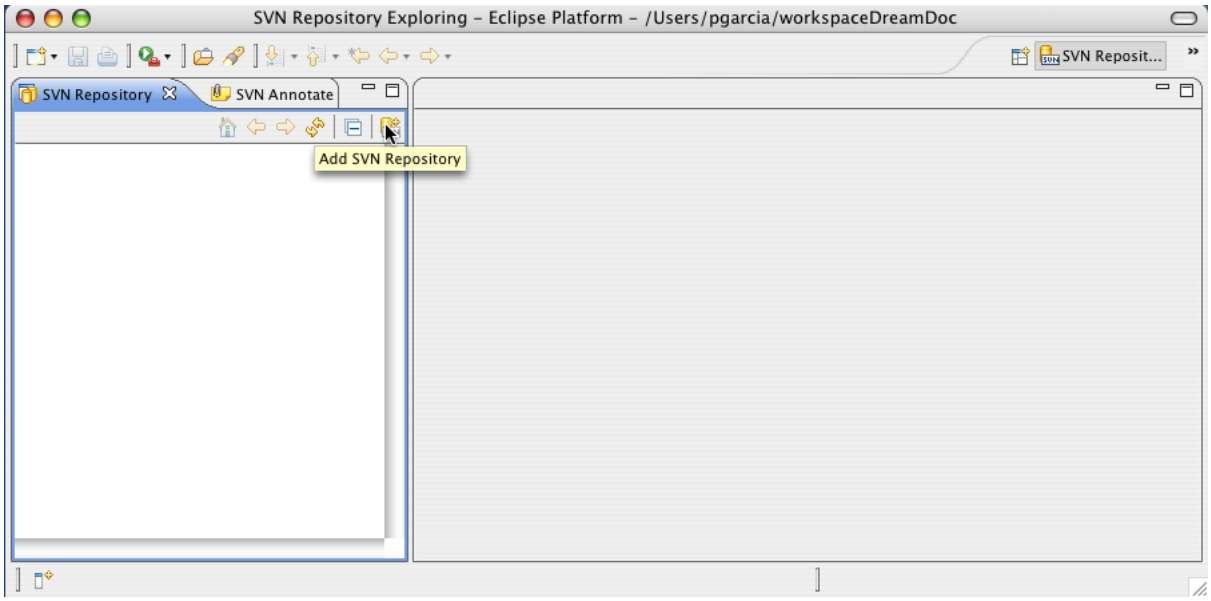

Figure 1: Adding a SVN repository.

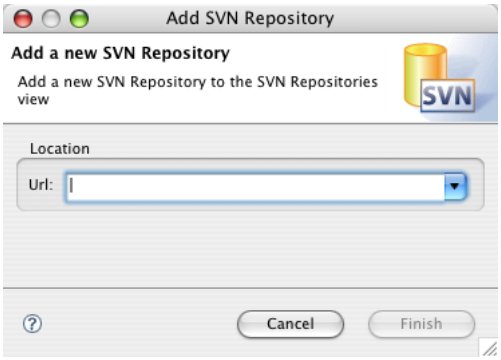

Figure 2: "Add a new SVN Repository" dialog box.

- In the "SVN Repositories" view, right click on the `trunk` directory, select `Check Out` (see figure 3).

A new project, called *trunk*, has been created. You can browse it using the "Java" perspective.

1.4 Configuring Eclipse

1.4.1 Configuring the CheckStyle plugin

The CheckStyle plugin is configured as follows:

- Open the `Preferences` dialog box (`Window` > `Preferences`).

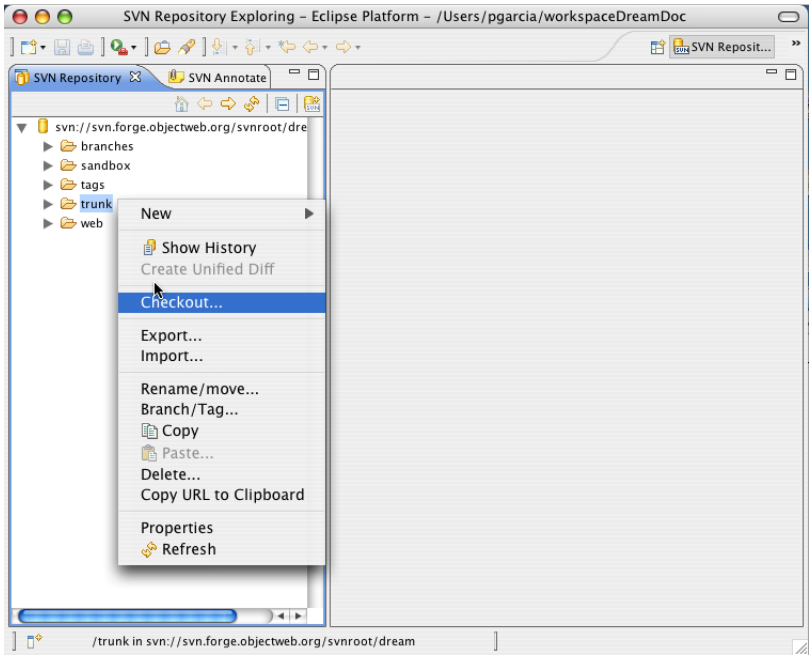

Figure 3: Checkout trunk module

- Select `CheckStyle` in the list (if it is not present, see section 1.1 to install the plugin).
- Click on the `New` button to add the DREAM configuration.
- Fill in the dialog box (see figure 4).

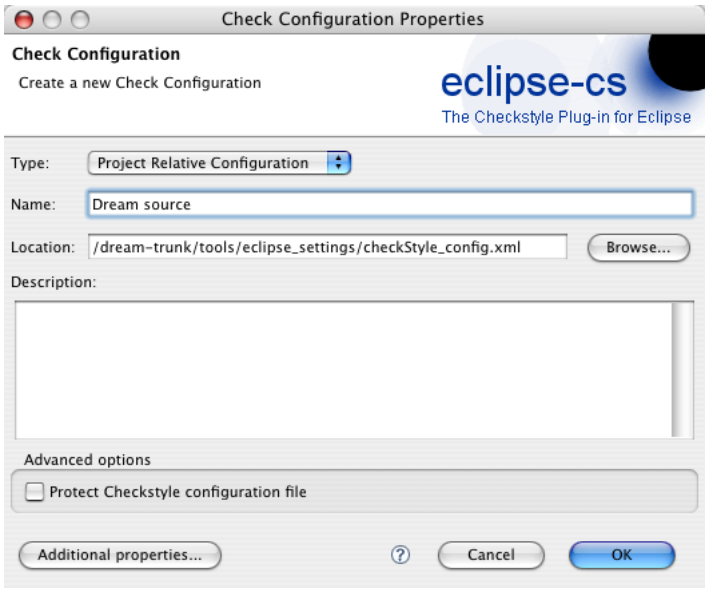

Figure 4: CheckStyle configuration

1.4.2 Configuring the code formatter

The code formatter is a code beautifier. It must be configured as follows:

- Open the `Preferences` dialog box (`Window` > `Preferences`).
- Select `Java` > `Code Style` > `Code Formatter` in the list (see figure 5).

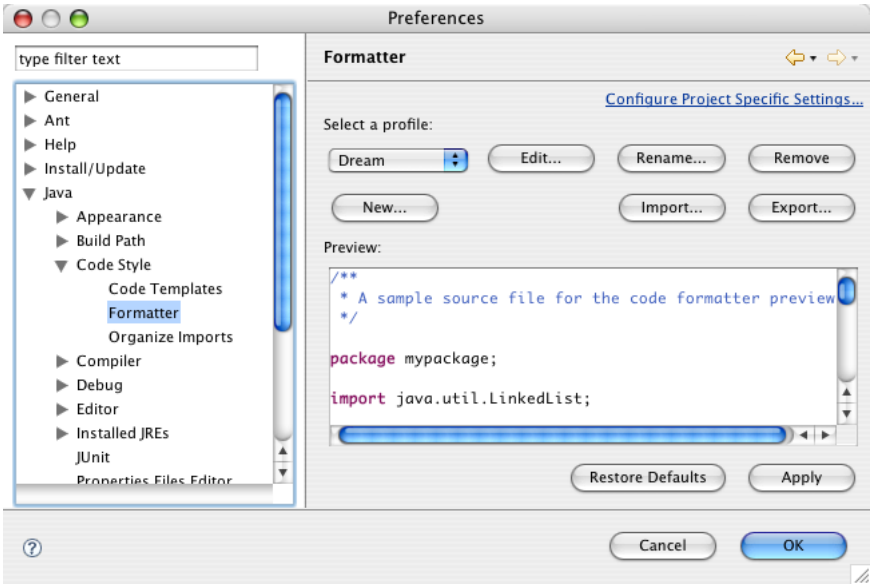

Figure 5: Code formatter preferences

- Click on the `Import` button.
- Go into your workspace directory.
- Select the `code_formater.xml` file in the `tools/eclipse_settings` directory.

1.4.3 Using the code formatter

The code formatter can be used when editing Java source files with the following shortcut: `Ctrl-Shift-F`.

2 Inside Dream modules

2.1 Module structure

2.1.1 Multi-Module Maven Project

Each multi-module project has the same structure as shows the picture 6.

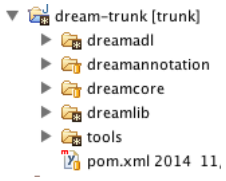

Figure 6: Multi-Module Directory Layout.

- The `pom.xml` file which contains a detailled description of the multi-module project and in particular the list of sub-modules.
- Directories corresponding to sub-modules. On the pictures 6, the sub-modules are `dreamadl`, `dreamannotation`, `dreamcore` and `dreamlib`.

2.1.2 Single Module Maven Project

Each Dream module has approximately the same structure as shows the picture 7. It is conform to maven standard directory layout.

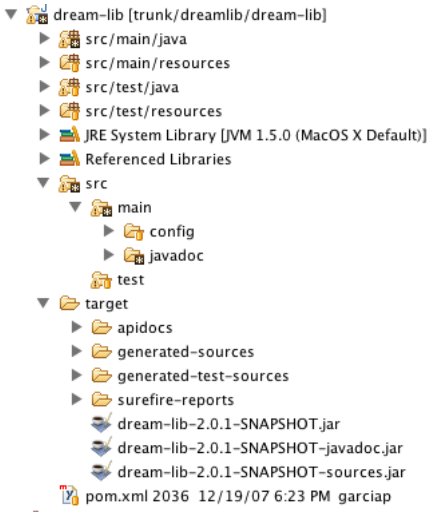

Figure 7: Single-Module Directory Layout.

- The `src/main/java` directory contains java sources of the module.

- The `src/main/resources` directory contains other resources of the module. These resources are mostly fractal ADL files.
- The `src/test/java` directory contains java sources of junit tests.
- The `src/test/resources` directory contains other resources of junit tests. These resources are mostly fractal ADL files.
- The `src/main/javadoc` directory contains files used to generate javadoc of the module.
- The `src/main/config` directory contains configuration files (ex : monolog.properties to configure logger).
- Every file generated during the maven build process are put inside the `target` directory. It contains :
  - a `generated-sources` directory containing java source files generated by spoon.
  - a `classes` directory containing compiled classes and fractal files generated by spoon.
  - an `apidocs` directory containing generated javadoc.
  - a `test-classes` directory containing compiled java classes.
  - a `surefire-reports` directory containing reports of junit tests.
  - 3 jar files :
    - *<artifactId>*-*<version-number>*.jar : this jar contains compiled classes and fractal ADL definitions.
    - *<artifactId>*-*<version-number>*-sources.jar : this jar contains sources.
    - *<artifactId>*-*<version-number>*-javadoc.jar : this jar contains generated javadoc.

2.2 Make a distribution and run the examples

TODO

3 HelloWorld example

This section describes how to write a simple "HelloWorld" application with DREAM.

The example used throughout this tutorial is a very simple application made of two primitive user components encapsulated by a composite component (see the figure 8). The first primitive component is a *message* producer: an internal activity produces messages containing a string, and pushes them on the component's `push` client interface. The second primitive component is a message consumer: it provides a `push` interface. When this component receives a message, it prints on the console the string contained in the message and then deletes the message. These two components have a messagemanager client interface since they create and delete messages. Finally the producer component has a `org.objectweb.dream.control.activity.manager.TaskManager` client interface in order to be able to execute tasks.

These two components can be bound directly (primitive binding) or using various components from the DREAM library (composite binding).

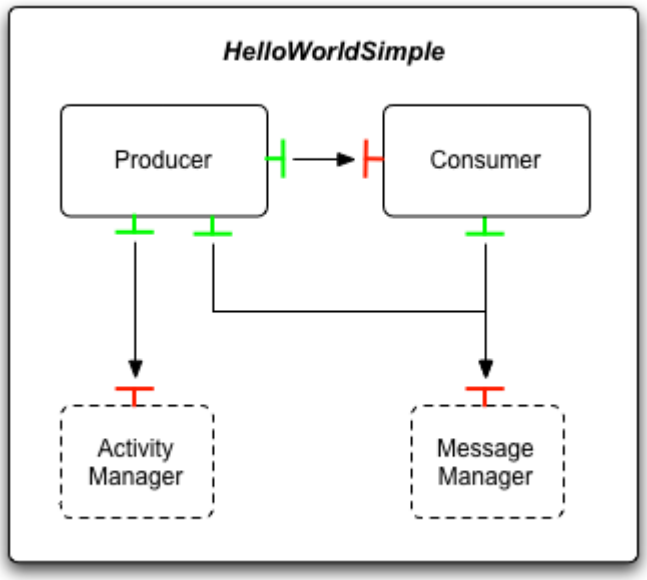

Figure 8: Simple Helloworld Example.

In this section we explain how to write the components described above and how to assemble and run them using the fractal ADL. The code and the ADL files described here can be found in the `trunk/dreamlib/examples/helloworld` directory of the dreamlib module.

3.1 Implementation

The interface used to exchange messages between client and server components is the `org.objectweb.dream.Push` interface of the dreamcore module.

3.1.1 The `HelloWorldChunk` implementation

In our example, producer and consumer components exchange messages containing a string. Since DREAM messages are composed of *chunks*, we need to write a class extending the `org.objectweb.dream.message.AbstractChunk` class and containing a String field. The code of this class is the following:

```
public class HelloWorldChunk extends AbstractChunk<HelloWorldChunk> {
  public static final String DEFAULT_NAME     = "helloworld-chunk";
  private String             message;

  public final String getMessage() {
    return message;
  }

  public final void setMessage(String message) {
    this.message = message;
  }

  // Chunk Interface Implementation

  protected HelloWorldChunk newChunk() {
    return new HelloWorldChunk();
  }

  protected void transfertStateTo(HelloWorldChunk newInstance) {
    newInstance.message = message;
  }

  public void recycle() {
    message = null;
  }

  // Externalizable Interface Implementation

  public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
    message = in.readUTF();
  }

  public void writeExternal(ObjectOutput out)
      throws IOException {
    out.writeUTF(message);
  }
}
```

- The `Chunk` interface is used by the associated chunk factory which manages chunk creation/destruction/recycling...
- The `Externalizable` interface is used by message codecs when the message is serialized.

3.1.2 The consumer implementation

The consumer component receives message on its `Push` server interface, print it using the logger and delete it. Here is the code of the class :

```
@DreamComponent
@Provides(interfaces={@Interface(name="in-push",signature=Push.class)})
public class Consumer implements Push
{
  @Service
  Component            ref;
  @DreamMonolog
  Logger               logger;
  @Attribute(argument = "nbMessageToReceive")
  int                  nbMessageToReceive;

  // Client interfaces
  @Requires(name = "message-manager")
  MessageManagerType messageManagerItf;

  // Implementation of the Push interface
  public void push(Message message) throws PushException
  {
    HelloWorldChunk c = (HelloWorldChunk) messageManagerItf
        .getChunk(message, HelloWorldChunk.DEFAULT_NAME);
    if (c == null)
    {
      throw new PushException("Unable to find Helloworld chunk");
    }
    logger.log(BasicLevel.INFO, c.getMessage());
    messageManagerItf.deleteMessage(message);
    this.setNbMsgToReceive(--nbMessageToReceive);
  }

  public synchronized int getNbMsgToReceive()
  {
    return nbMsgToReceive;
  }
  public synchronized void setNbMsgToReceive(int nbMsgToReceive)
  {
    this.nbMsgToReceive = nbMsgToReceive;
  }
}
```

To lightweight the code, annotations (from `dream-annotation` and `fractal-spoonlet` modules) are used in java implementation of fractal components :

- `@DreamComponent` declares the component.
- `@Provides` declares server interfaces of the component. This component provide one `Push` interface where messages are received.

- `@Service` declares the reflexive reference to the component. This field is required to register the logger declared with the `@DreamMonolog` annotation
- `@Requires` declares client interfaces of the component. Here the `messageManagerItf` is needed to delete processed messages.
- `@Attribute` declares a fractal attribute for the consumer component. It represents the number of message that the consumer still wants to receive.

These annotations let the programmer write almost only functionnal code, i.e. the `Push` interface implementation.

3.1.3 The producer implementation

The producer component defines a periodic *task* which pushes a message containing a `HelloWorldChunk`. This component has two client interfaces: the `push` interface used to send produced messages via the `messageManagerItf` interface. It has no server interface. Here is the code corresponding to the component declaration :

```
/** Producer component implementation. */
@DreamComponent(controllerDesc = "activeDreamPrimitive")
public class Producer
{
  private ChunkFactoryReference<HelloWorldChunk> hcFactory;

  @Service
  private Component ref;
  @DreamMonolog
  private Logger logger;

  @Requires(name = "out-push")
  private Push                                    outPushItf;

  @Requires(name = "message-manager")
  private MessageManagerType                      messageManagerItf;

  @Requires(name = "task-manager")
  private TaskManager                             taskManagerItf;
```

Notes :

- the `TaskManager.ITF_NAME` is not used directly by the component implementation. It is used only by the task controller of the component at start time to register the task.

We now implement the inner task. A DREAM task implements the `org.objectweb.dream.control.activity.task.Task` interface. DREAM provides an abstract implementation of this interface. So to implement our task, we define an inner class extending `org.objectweb.dream.control.activity.task.AbstractTask`:

```
private class ClientTask extends AbstractTask
{
  private String                                message;
  int                                           seqno;
  ChunkFactoryReference<HelloWorldChunk> chunkFactory;

  public ClientTask(String message)
  {
    super(message);
    this.message = message;
    seqno = 0;
    chunkFactory = messageManagerItf
        .getChunkFactory(HelloWorldChunk.class);
  }

  public Object execute(Object hints) throws InterruptedException
  {
    try
    {
      Message msg = messageManagerItf.createMessage();
      HelloWorldChunk chunk = messageManagerItf
          .createChunk(chunkFactory);
      logger.log(BasicLevel.DEBUG, "pushing message containing "
          + message + " seqno=" + (seqno));
      chunk.setMessage(message + " seqno=" + (seqno++));
      messageManagerItf.addChunk(msg, HelloWorldChunk.DEFAULT_NAME,
                                 chunk);
      outPushItf.push(msg);
    }
    catch (Exception e)
    {
      e.printStackTrace();
      return STOP_EXECUTING;
    }
    return EXECUTE_AGAIN;
  }
}
```

The `execute` method creates a message and a `HelloWorldChunk`, sets the chunk's string, adds the chunk in the message and pushes the message.

To make this task executed, we need to register it with the *task controller*. This is done in the `beforeFirstStart` method. This method which is annotated with the `DreamLifeCycle(on=DreamLifeCycleType.FIRST_START)` is called the first time the component is started. The code of this method is the following :

```
@DreamLifeCycle(on = DreamLifeCycleType.FIRST_START)
protected void beforeFirstStart(Component componentItf)
    throws IllegalLifeCycleException
{
  try
  {
    Map<String, Object> hints = new HashMap<String, Object>();
    hints.put("period", new Long(1000));
    Util.addTask(componentItf,
      new ClientTask("Hello World task1"), hints);
    logger.log(BasicLevel.DEBUG, "tasks added");
  }
  catch (Exception e)
  {
    throw new IllegalLifeCycleException("Can't add task");
  }
}
```

The helloworld task is defined as periodic by adding a `"period"` key in the hints map. The unit of this period is millisecond.

3.2 ADL

In this section we will see the generated ADL of primitive components and how to write ADL files that allow binding these components.

3.2.1 The producer and consumer ADLs

As we have seen in 3.1.2 the consumer component provides a `push` interface and requires a `message manager` interface. So its generated ADL (in target/classes directory) is the following:

```
<definition name="helloworld.Consumer">
  <interface name="in-push" role="server"
    signature="org.objectweb.dream.Push"/>
  <interface name="message-manager" role="client"
    signature="org.objectweb.dream.message.MessageManager"/>
  <content class="helloworld.ConsumerImpl"/>
  <controller desc="dreamPrimitive"/>
</definition>
```

The generated ADL of the producer component is the following:

```
<definition name="helloworld.Producer">
  <interface name="out-push" role="client"
    signature="org.objectweb.dream.Push"/>
  <interface name="message-manager" role="client"
    signature="org.objectweb.dream.message.MessageManager"/>
  <interface name="task-manager" role="client" signature=
      "org.objectweb.dream.control.activity.manager.TaskManager"/>
  <content class="helloworld.ProducerImpl"/>
  <controller desc="activeDreamPrimitive"/>
</definition>
```

Note the `controller` element. Since the DREAM framework defines its own controllers (logger, activities ...), we need to specify the controller description. The available controller descriptors are:

- `dreamPrimitive` for simple primitive component
- `dreamUnstoppablePrimitive` for primitive component that cannot be stopped alone (without life cycle interceptor)
- `activeDreamPrimitive` for primitive component containing activities.
- `activeDreamUnstoppablePrimitive` for primitive component that cannot be stopped alone and containing activities
- `dreamComposite` for simple composite component
- `dreamUnstoppableComposite` for composite component that cannot be stopped alone

Note : the life cycle interceptor also blocks incoming calls on server interface until the component is started.

3.2.2 The simple HelloWorld ADL

In this ADL we describe a composite component containing the producer and consumer components, plus two other required components (message manager, activity manager). The producer is bound directly to the consumer:

```
<definition name="helloworld.HelloWorldSimple">
  <component name="Producer" definition="helloworld.Producer(5)"/>
  <component name="Consumer" definition="helloworld.Consumer"/>
  <component name="MessageManager"
    definition="org.objectweb.dream.message.MessageManager"/>
  <component name="ActivityManager"
    definition="org.objectweb.dream.control.activity
.manager.ActivityManager"/>
  <binding client="Producer.out-push" server="Consumer.in-push"/>
  <binding client="Producer.message-manager"
    server="MessageManager.message-manager"/>
  <binding client="Consumer.message-manager"
    server="MessageManager.message-manager"/>
  <binding client="Producer.task-manager"
    server="ActivityManager.task-manager"/>
  <controller desc="dreamComposite"/>
</definition>
```

We can execute this application by running `mvn` in the `trunk/dreamlib/examples/helloworld` directory. The application is deployed and started by the Fractal ADL factory. This results in a message being printed on the console every second:

```
  [java] [INFO] thread=10 helloworld.HelloWorldSimple.Consumer.impl
ConsumerImpl.push: Hello World task1 seqno=0
  [java] [INFO] thread=10 helloworld.HelloWorldSimple.Consumer.impl
ConsumerImpl.push: Hello World task1 seqno=1
  [java] [INFO] thread=10 helloworld.HelloWorldSimple.Consumer.impl
ConsumerImpl.push: Hello World task1 seqno=2
  [java] [INFO] thread=10 helloworld.HelloWorldSimple.Consumer.impl
ConsumerImpl.push: Hello World task1 seqno=3
```

3.2.3 More sophisticated HelloWorld

We now add some components from the DREAM component library to build a composite binding from the message producer to the consumer. The first component we add is a *Queue*. We choose to add a "Push/Push active queue", this queue provides a push interface used to add incoming messages. These messages are then pushed on a push client interface by the internal activity of the queue. The ADL describing our new application is the following:

```
<definition name="helloworld.HelloWorldQueue"
  extends="helloworld.HelloWorldSimple">
  <component name="Queue"
    definition="org.objectweb.dream.queue.
____PushPushQueueActive(10,ExceptionThrowing,FIFO,1)">
    <component name="ActivityManager" definition="./ActivityManager" />
    <component name="MessageManager" definition="./MessageManager" />
  </component>
  <binding client="Producer.out-push" server="Queue.in-push" />
  <binding client="Queue.out-push" server="Consumer.in-push" />
  <controller desc="dreamComposite" />
</definition>
```

This ADL extends the previous one. It adds the `Queue` component with the specified definition and arguments. This component is a composite component, it must contain an activity manager and a message manager, so the activity manager and the message manager defined in the `HelloworldSimple` ADL are shared with this composite. Then the binding from the producer to the consumer is overridden and the producer is bound to the `in-push` Queue interface and the `out-push` interface of the Queue is bound to the message consumer.

To run this application you can executing the following command:

```
mvn -Prun -Dcom=Queue
```

The output on the console will not differ from the previous example since only the consumer logs messages on the console. To see the queue's debug messages uncomment the following line in the `src/main/resources/monolog-dreamlib-helloworld.monolog.properties` file :

```
#logger.helloworld.HelloWorldQueue.level  DEBUG
```

This line sets the debug level to loggers of the Queue component. Loggers are named using the architecture of the application. The `Queue` component is encapsulated in the `helloworld.HelloWorldQueue` composite, which is the root component, so each logger of the queue component (controller's logger and sub-components' logger) has a name beginning with `helloworld.HelloWorldQueue.Queue`

A Dream component can have multiple loggers for controllers and implementation. Each one can have a specific level defined in the `monolog.properties` file. For instance if you want to activate the debug level of the logger of the life cycle controller of the consumer component you add the following line to the `monolog.properties` file:

```
logger.helloworld.HelloWorldQueue.Consumer.life-cycle.level DEBUG
```

We now add channel components between the producer and the consumer. Channel components allow to exchange messages between different address spaces. Various channel implementations are provided by the Dream library. In this example, we use the TCP based channel.

- The `TCPChannelOut` component allows to send messages to a given destination using a TCP socket. This destination is found in each outgoing message as a `ExportIdentifierChunk`. This chunk contains the hostname and the listening port of the destination. It can be added to outgoing messages by the `AddIPExportIdChunk` component.

- The `TCPChannelInOut` component can receive incoming messages sent by remote `TCPChannelOut` components. In addition, this compoent can also send outgoing messages as the `TCPChannelOut` component.

In our example we deploy both ChannelIn and ChannelOut in the same JVM. The ADL describing our new application is the following:

```
<definition name="helloworld.HelloWorldTCPChannel"
  extends="helloworld.HelloWorldSimple">

  <component name="AddDestination" definition=
"org.objectweb.dream.channel.AddIPExportIdChunk(destination,
localhost,1600)"/>

  <component name="ChannelOut" definition=
"org.objectweb.dream.channel.ChannelOutTCPStack(destination,true,2)">
    <component name="MessageManager" definition="./MessageManager"/>
    <component name="ActivityManager" definition="./ActivityManager"/>
    <component name="MessageCodec" definition=
"org.objectweb.dream.message.codec.MessageCodecObjectStream(false)">
      <component name="MessageManager" definition="./MessageManager"/>
    </component>
  </component>

  <component name="ChannelIn" definition=
"org.objectweb.dream.channel.ChannelInOutTCPStack(destination,true,
null,2,1600,localhost)">
    <component name="MessageManager" definition="./MessageManager"/>
    <component name="ActivityManager" definition="./ActivityManager"/>
    <component name="MessageCodec" definition=
"org.objectweb.dream.message.codec.MessageCodecObjectStream(false)">
      <component name="MessageManager" definition="./MessageManager"/>
    </component>
  </component>

  <binding client="Producer.out-push"
    server="AddDestination.in-push"/>
  <binding client="AddDestination.out-push"
    server="ChannelOut.in-push"/>

  <binding client="ChannelIn.out-push"
    server="Consumer.in-push"/>
</definition>
```

This ADL extends the previous one. It adds tree components:

- the `AddDestination` component add a `ExportIdentifierChunk` in every messages pushed by the producer. The name of the added chunk, hostname and port are specified as ADL arguments (in our case `"destination"`, `"localhost"` and `1600`).
- the `ChannelOut` composite component is responsible for sending messages created by the producer. The first argument (`"destination"` here) specifies the name of the chunk that contains the identifier of the destination of the message. The second one specifies if this chunk should be removed before the message is sent. As we seen in the queue example, this composite component shares `MessageManager` and `ActivityManager` components. It also contains a `MessageCodec` component. This latter is responsible for serializing messages.
- the `ChannelIn` composite component accept incoming connections on the port 1600 (as it is specified as argument) and push received messages to the consumer.

To run this application you can executing the following command:

```
mvn -Prun -Dcom=TCPChannel
```

An UDP example with the same architecture can be launched with the following command :

```
mvn -Prun -Dcom=UDPChannel
```