

# How to use DynaBioS-2-Java-platform

## Contents

1. Introduction.....	3
2. System overview.....	4
3. System functions.....	5
4. Component handler type.....	6
4.1. Internal component handler.....	6
4.2. Remote component handler.....	6
4.3. User-defined handler.....	7
5. System-defined classes.....	8
5.1. Class Main.....	8
5.2. Class SimulationScenario.....	8
5.2.1. initialize.....	9
5.2.2. terminate.....	9
5.2.3. preProcess.....	10
5.2.4. mainProcess.....	10
5.2.5. eventProcess.....	10
5.2.6. postProcess.....	11
5.3. Class SimulationModel.....	11
5.4. Class Component.....	12
5.4.1. processMessage.....	12
5.5. Class ComponentMessage.....	12
5.6. Class Main for the remote component server.....	13
6. Classes used in DynaBioS-2-Java-platform framework.....	14
6.1. Class ComponentEntry.....	14
6.2. Class SystemLoader.....	14
6.2.1. load.....	14
6.3. Class Message.....	14
6.3.1. makeMessage.....	15
6.3.2. getComponentID / setComponentID.....	15
6.3.3. getDataSize.....	15
6.3.4. getCommand / setCommand.....	15
6.3.5. getData / setData.....	15
6.4. Class MessageData.....	15
6.5. Class RemoteComponentServer.....	16
6.5.1. main.....	16

# How to use DynaBioS-2-Java-platform

7.	Classes of the DynaBioS-2-Java-platform core.....	17
7.1.	Class SystemCore .....	17
7.1.1.	construct.....	17
7.1.2.	execute.....	17
7.2.	Class ComponentManager.....	18
7.2.1.	allocateComponent .....	18
7.2.2.	GetComponent .....	18
7.2.3.	releaseComponent.....	18
7.3.	Class MessageRouter.....	18
7.4.	Class SimulationController.....	19
7.4.1.	initialize .....	19
7.4.2.	terminate.....	19
7.4.3.	execute.....	19
7.4.4.	postMessage .....	19
7.4.5.	sendMessage .....	19
7.4.6.	processMessage.....	19
7.5.	Class ModelManager .....	20
7.5.1.	initialize .....	20
7.5.2.	terminate.....	20
7.5.3.	getModel.....	20
7.6.	Class ComponentHandler.....	20
7.6.1.	GetComponentID.....	20
8.	Example code.....	21
8.1.	Internal component handler.....	21
8.1.1.	Class HelloWorld.....	21
8.1.2.	Class HelloWorldModel .....	22
8.1.3.	Class HelloWorldScenario .....	22
8.1.4.	Class HelloWorldComponent.....	24
8.1.5.	Class HelloWorldComponentMessage .....	25
8.2.	Remote component handler.....	25
8.2.1.	Class HelloWorld.....	26
8.2.2.	Class HelloWorldRemoteServer .....	26

# How to use DynaBioS-2-Java-platform

## 1. Introduction

DynaBioS-2-Java-platform is used to, in a simple way, construct a system with a controller and one or more units. Each unit can run internally in the controller or from a remote server, which then use TCP/IP to communicate with the controller. It will envelop the message and send data as binary without the user having to take care about the conversion. Instead the user uses the available put- and get-methods to write and read the message data of different data types.

DynaBioS-2-Java-platform has a component based architecture made in Java. It has a set of abstract classes and abstract methods (user process) that the user creates subclasses of to make it easy to call DynaBioS-2-Java-platforms system functions. It uses a scenario to define the systems behavior, a model to define the data and one or more components to define each unique unit in the system.

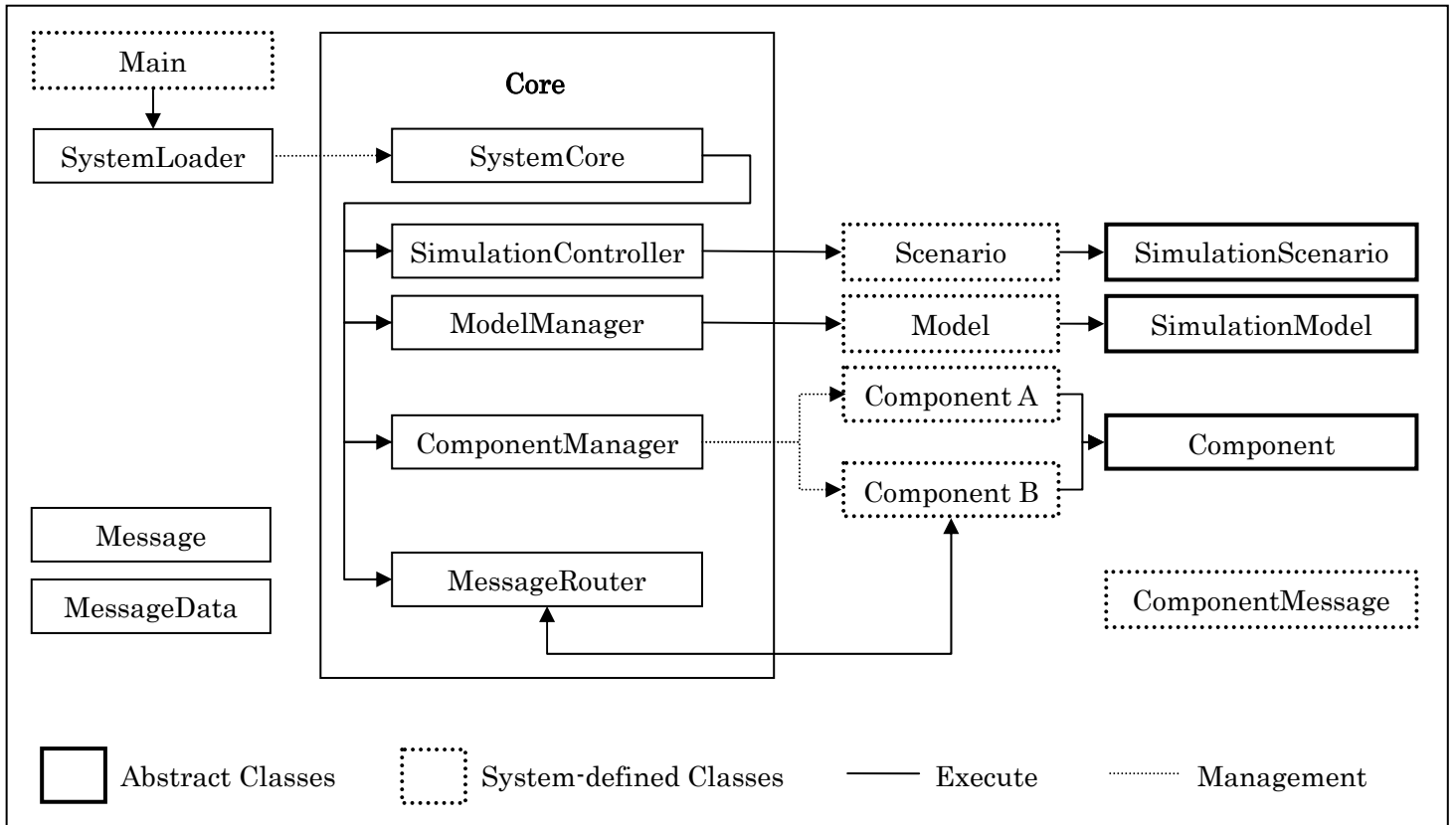
This description is for users who will use the DynaBioS-2-Java-platform for their own system. It is not meant to be a full description of the code. If you need more information, then see the Javadoc.

This document describes how to use the DynaBioS-2-Java-platforms API. It describes the system-defined classes the user needs to create to use the DynaBioS-2-Java-platform. Some of these classes are subclasses of DynaBioS-2-Java-platform classes, in which case user processes and system functions are also described. It describes classes used directly by the users. To understand the data flow it also describes classes and methods that the user need to know. In the end of the document you will find two examples of code using DynaBioS-2-Java-platform. The first example is for a controller with an internal component handler and the second example does the same thing, but by using a remote component handler instead.

# How to use DynaBioS-2-Java-platform

## 2. System overview

This figure shows an overview of the system of DynaBioS-2-Java-platform. All system-defined classes in the figure are described in chapter 5, all classes used directly by the users are described in chapter 6 and in chapter 7 you will find the classes you need to be familiar with to understand the data flow.



# How to use DynaBioS-2-Java-platform

## 3. System functions

DynaBioS-2-Java-platform has some system functions which can be called from the abstract methods (user processes) that are implemented in your Scenario.

When calling system functions from your Scenario, first the class SimulationScenario will be called and by using Java super method-call an internal instance of class SystemCore can be called and by use of the created instance made in the method **construct**, can your Scenario, Model or Component execute the called method.

The following system functions can be called from your Scenario:

- The method **getModel** for class ModelManager can be called. (See chapter 7.5).
- The methods **allocateComponent**, **getComponent** and **releaseComponent** for class ComponentManager can be called. (See chapter 7.2).
- The methods **sendMessage**, **postMessage** and **processMessage** for class SimulationController can be called. (See chapter 7.4).

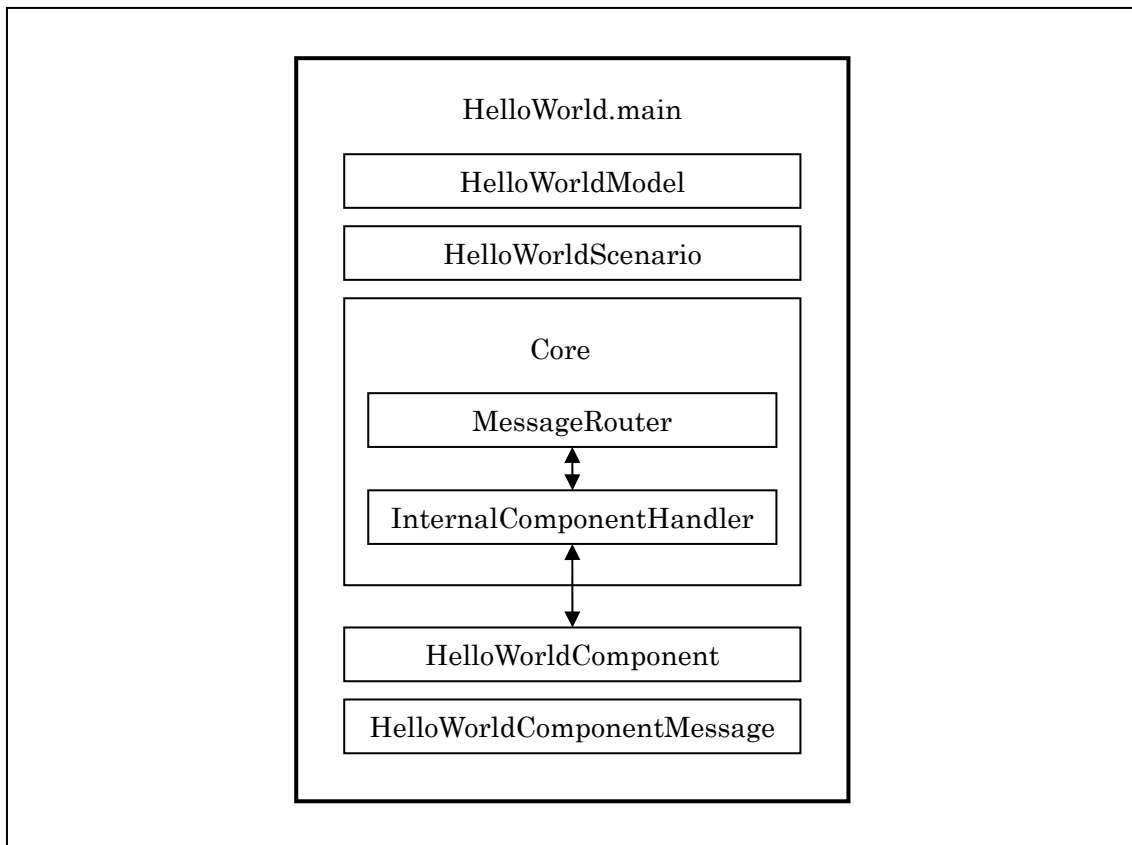
# How to use DynaBioS-2-Java-platform

## 4. Component handler type

The component handler type can be internal, remote or user-defined. The differences between component handler types are described in this chapter.

### 4.1. Internal component handler

When using an internal component handler, the component runs on the same computer as the main program. The following figure shows an overview of the example in chapter 8.1, when an internal component handler is used. The InternalComponentHandler use method-call when communicate with HelloWorldComponent.

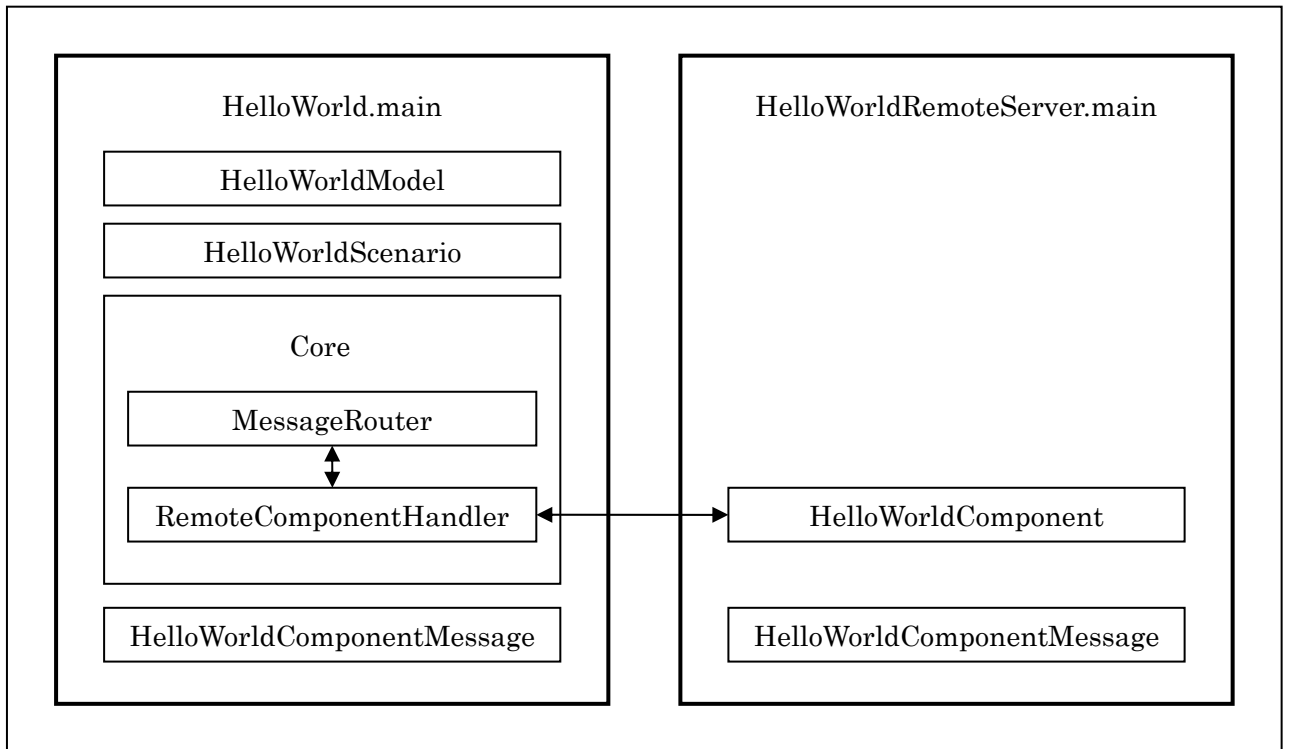


### 4.2. Remote component handler

A remote component handler makes it possible to send data between a controller and one or more remote component servers over TCP/IP. To set up and run a remote component server, use the class RemoteComponentServer (see chapter 6.5). The

## How to use DynaBioS-2-Java-platform

following figure shows an overview of the controller and the remote component server as described in the example in chapter 8.2, when the remote component handler is used. The RemoteComponentHandler use TCP/IP when communicating with HelloWorldComponent.



### 4.3. User-defined handler

The user-defined handler is not described in this document.

# How to use DynaBioS-2-Java-platform

## 5. System-defined classes

The user is required to construct a minimum of 4 classes. These are; the Main class for the controller, your own system-defined subclass of the class SimulationScenario, here after referred to as Scenario, your own system-defined subclass of the class SimulationModel, here after referred to as Model and your own system-defined subclass of the class Component, here after referred to as your Component. It is also common to create a class that contains methods for all system-defined message types, here after referred to as ComponentMessage.

In addition, if the component handler type is remote, then the user is required to construct a Main class for the remote component server.

This chapter describes all system-defined classes the user need to construct and what each of these classes does. User processes and common system functions to call from the Scenario, the Model and your Component are also described.

### 5.1. Class Main

Create a system-defined Main class.

This class makes an instance of ComponentEntry for your system (see chapter 6.1), an instance of the Scenario and an instance of the Model. Call the class SystemLoaders method **load** (see chapter 6.2), with the above objects as input argument.

### 5.2. Class SimulationScenario

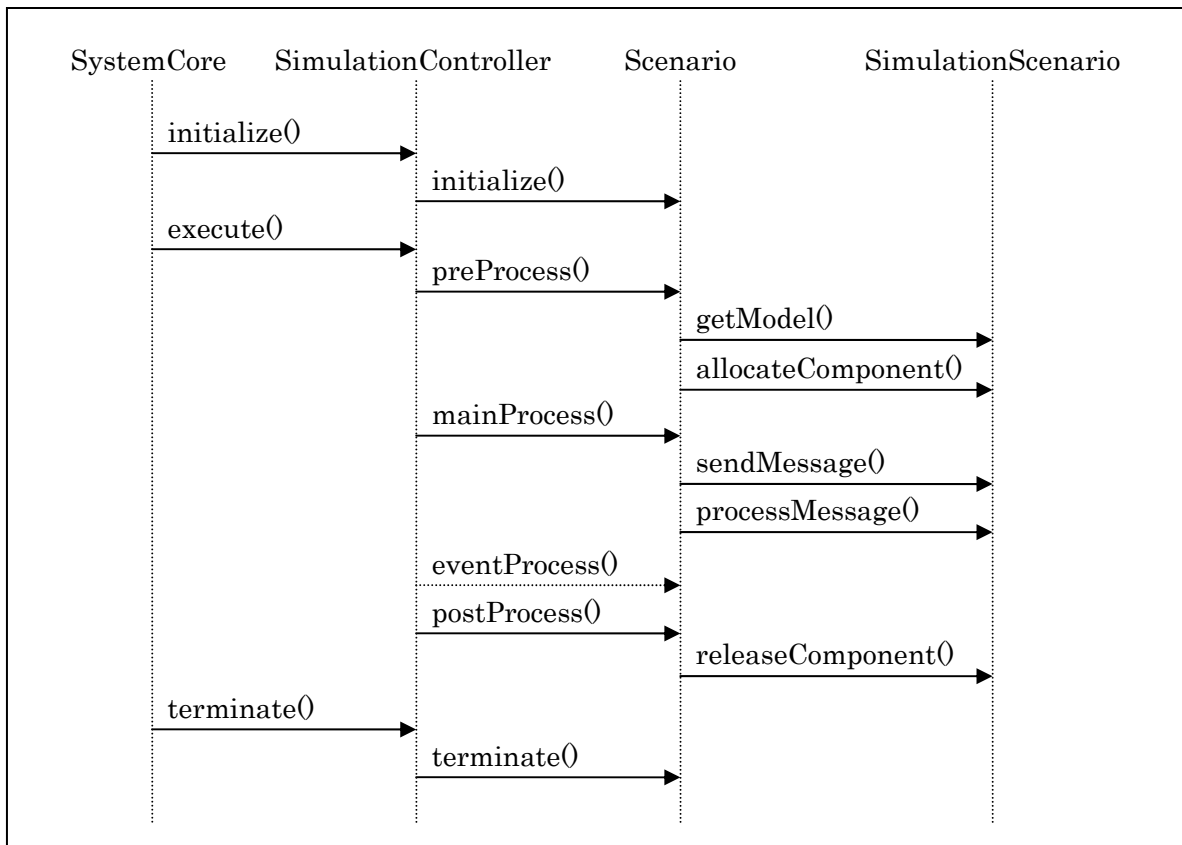
Create a system-defined subclass of the class SimulationScenario for your Scenario.

The SimulationScenario class is the abstract super class of all simulation scenarios. It contains many user processes and system functions. For more information on the system functions, see chapter 3.

The most common system functions to call from your Scenario are described under each user process in this chapter. If no system function is described for a method, then it is not common to call a system function from this method.

## How to use DynaBioS-2-Java-platform

The following figure shows an overview of function calls for a common Scenario. Common system functions to call from your user processes are printed in the figure. For example, it is common to call **getModel** and **allocateComponent** from user process **preProcess**. In chapter 3 you can see what each system function will do.



### 5.2.1. initialize

Initialize the scenario. Observe that no system function can be called from here, because class SystemCore is not finished with its initiation.

### 5.2.2. terminate

Terminate the scenario. Observe that no system function can be called from here, because class SystemCore has already started to terminate.

# How to use DynaBioS-2-Java-platform

## 5.2.3. preProcess

A common system function to call is **getModel**, this will acquire your Model. This makes it possible to call the methods in your Model from the called class.

Another common system function to call is the **allocateComponent**, this will allocate a component and return the component handler. If the handler type is internal, then it will be an instance of your Component.

## 5.2.4. mainProcess

Make an instance of the class Message.

Call any of the class ComponentMessage methods to create a message that later can be sent from this method.

A common system function to call is the **sendMessage**, which synchronously sends a message (This method blocks control until the destination component processes the message). Another system function to send a message with is the **postMessage**, which asynchronously posts a message (This method returns control without blocking). DynaBioS-2-Java-platform will find the right Component by reading the Message component id and the user process **processMessage** (process a received message) for that Component will be called and executed (see chapter 5.4.1)

To process a received message the system function **processMessage** (request to process a received message) is used. The system will wait until a message is received from any of your Component's, and then the user process **eventProcess** for your Scenario will be called to process the received message.

When using **sendMessage**, **postMessage** and **processMessage** the message will be enveloped by DynaBioS-2-Java-platforms class NetworkMessageEnvelope. This will convert the message to/from a byte array. However this is not a part of this description since the user don't need to bother about the conversion.

## 5.2.5. eventProcess

## How to use DynaBioS-2-Java-platform

In this method it is common to check if the received message component id matches any of the allocated component handlers component id. This is done by calling the method **getComponentID** for class `Message` (see chapter 6.3) and by calling method **getComponentID** for class `ComponentHandler` (see chapter 7.6). If it matches, then check if there are any user defined commands in your class `ComponentMessage` that matches the received message command. This is done by calling method **getCommand** for class `Message`. If it matches, then the received message will be processed.

### 5.2.6. **postProcess**

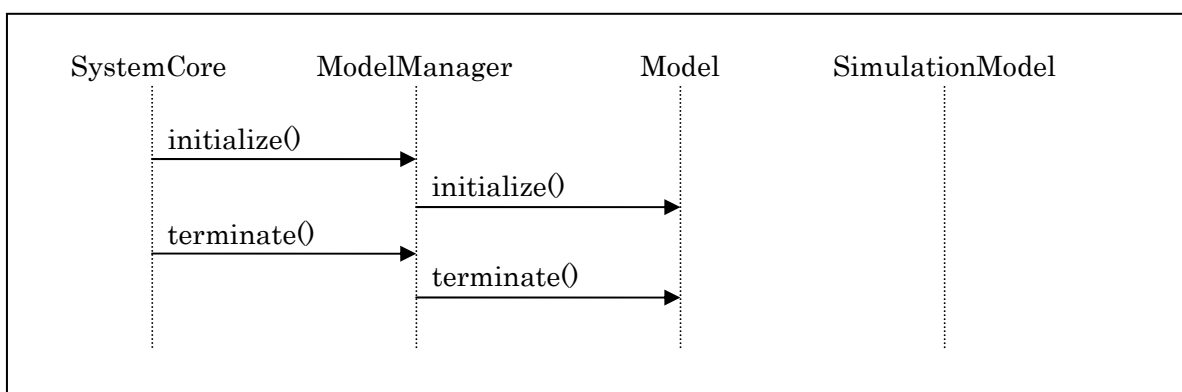
A common system function to call is **releaseComponent**, this will release your instance of `ComponentHandler`.

### 5.3. Class **SimulationModel**

Create a system-defined subclass of the class `SimulationModel` for your `Model`.

`SimulationModel` class is the abstract super class of all simulation models. It contains two user processes, **initialize** and **terminate**. This will initialize and terminate the model.

It is not common to call any system functions from your `Model` from these two methods as you can see in the following figure showing an overview of function calls for a common `Model`.



# How to use DynaBioS-2-Java-platform

## 5.4. Class Component

Create a system-defined subclass of the class Component for your Component.

The component class is the abstract super class of all simulation components. It contains tree user processes. It is not common to call any system function from the user processes, **initialize** and **terminate**. These will initialize and terminate the component.

### 5.4.1. processMessage

In this method it is common to check if the received message command matches to any of the user defined commands in your class ComponentMessage, by calling method **getCommand** for class Message. If it matches, then send a response message by calling the method **postMessage** with a response command. Response message have no data. After the system function **processMessage** (request to process a received message) have been called from your Scenario and the response message been received the user process **eventProcess** for your Scenario will be called to process the received message.

## 5.5. Class ComponentMessage

It is common to create a class that contains all system-defined message commands and methods for writing and reading these message types.

This class is used to write messages to an instance of class Message that later can be send by calling the system functions **sendMessage** or **postMessage** from your Scenario. To write the message, make an instance of class MessageData and use one or more available **put** methods to write the message data. Then call the method **makeMessage** for class Message to make the message.

Create methods for received system-defined message types. Use these methods to read the message data from your class Component's user process **processMessage** and/or from your Scenario's user process **eventProcess**. To read the message data call the method **getData** for class Message and use one or more available **get** methods for class MessageData, depending on data type.

### 5.6. Class Main for the remote component server

If the component handler type is remote, then the user is required to construct a Main class for the remote component server. This class should call the class RemoteComponentServers method **main** (see chapter 6.5), with the input argument for your Component class, binding address for the controller and port number. See the example in chapter 8.2 for more information on the input arguments.

# How to use DynaBioS-2-Java-platform

## 6. Classes used in DynaBioS-2-Java-platform framework

Here are descriptions of all classes and their methods that are used directly by the user.

### 6.1. Class ComponentEntry

Save all components used in your system as an array of this class.

Class ComponentEntry has the following private variables.

ComponentEntry	string	componentID	ID name of a component
	int	handlerType	Type of handler
	int	maxAllocation	Max number of allocatable handlers
	string	className	Class name of the component if type is internal
	string[]	options	Option strings

See the examples in chapter 8 for more information on how to set data for internal or remote component handler types.

### 6.2. Class SystemLoader

#### 6.2.1. load

This method loads the system. This means that it makes an instance of class SystemCore (see chapter 7.1) and calls the method **construct** with input argument for your Scenario and Model and then calls the method **execute** for the object of ComponentEntry.

### 6.3. Class Message

Class Message has the following private variables.

Message	int	componentID	Component id
	int	dataSize	Data size
	int	command	Command

## How to use DynaBioS-2-Java-platform

	MessageData	data	Data, byte buffer
--	-------------	------	-------------------

### 6.3.1. **makeMessage**

Use the class Message own methods to set all private variables for this class.

### 6.3.2. **getComponentID / setComponentID**

These methods are used to return and set the message component id.

### 6.3.3. **getDataSize**

This method is used to return the message data size.

### 6.3.4. **getCommand / setCommand**

These methods are used to return and set the message command.

### 6.3.5. **getData / setData**

These methods are used to return and set the message data.

## 6.4. **Class MessageData**

Class MessageData save the data in a byte buffer. To read and write data from the byte buffer for different data types, the following methods are available:

```
public final byte getByte()
public final char getChar()
public final double getDouble()
public final float getFloat()
public final int getInt()
public final long getLong()
public final short getShort()
public final String getString()
public final byte[] getByteArray(final int length)
```

## How to use DynaBioS-2-Java-platform

```
public final double[] getDoubleArray()
public final double[] getDoubleArray(final int length)
public final void getDoubleArray(final double[] doubleArray)
public final void putByte(final byte b)
public final void putChar(final char value)
public final void putDouble(final double value)
public final void putFloat(final float value)
public final void putInt(final int value)
public final void putLong(final long value)
public final void putShort(final short value)
public final void putString(final String value)
public final void putByteArray(final byte[] src)
public final void putDoubleArray(final double[] src)
public final byte[] toRawData() {
final ByteBuffer getByteBuffer() {
final void putByteBuffer(final ByteBuffer buf) {
```

### 6.5. Class RemoteComponentServer

#### 6.5.1. main

This method executes the remote component server. Input argument for this method is your Component class, binding address for the controller and port number. See the example in chapter 8.2 for more information on the input arguments.

# How to use DynaBioS-2-Java-platform

## 7. Classes of the DynaBioS-2-Java-platform core

Here are descriptions of all classes and their methods that the user needs to know about to understand the data flow.

### 7.1. Class SystemCore

#### 7.1.1. **construct**

This method creates instances of the class SimulationController for your Scenario and of the class ModelManager for your Model, the ComponentManager and the MessageRouter.

#### 7.1.2. **execute**

This is the core of DynaBioS-2-Java-platform. From here, your own system-defined classes will be called.

Execute will call these methods in this order:

- The method **initialize** for the object componentManager (see chapter 7.2), this will make it so that the method **initialize** for your Component will be called when the system function **allocateComponent** is called.
- The method **initialize** for the object messageRouter (see chapter 7.3).
- The method **initialize** for the object simulationController (see chapter 7.4), this will call your Scenario's method **initialize**.
- The method **initialize** for the object modelManager (see chapter 7.5), this will call your Model's method **initialize**.
- The method **registerComponentEntries** for the object componentManager (see chapter 7.2) with input argument for your own system-defined ComponentEntry.
- The method **execute** for the object simulationController (see chapter 7.4), this will

# How to use DynaBioS-2-Java-platform

run your Scenario's methods **preprocess**, **mainProcess** and **postProcess**.

- The method **terminate** for the object modelManager (see chapter 7.5), this will call your Model method **terminate**.
- The method **terminate** for the object simulationController (see chapter 7.4), this will call your Scenario's method **terminate**.
- The method **terminate** for the object messageRouter (see chapter 7.3).
- The method **terminate** for the object componentManager (see chapter 7.2), this will make it so that the method **terminate** for your Component will be called when the system function **releaseComponent** is called.

## 7.2. Class ComponentManager

Class ComponentManager, manager all allocated Components.

### 7.2.1. allocateComponent

Allocates a component handler. This method is called by the system function with the same name.

### 7.2.2. getComponent

Returns the component handler by using the component id. This method is called by the system function with the same name.

### 7.2.3. releaseComponent

Release a component handler. This method is called by the system function with the same name.

## 7.3. Class MessageRouter

Class MessageRouter will, by using class ComponentManager and system function

## How to use DynaBioS-2-Java-platform

**GetComponent**, do so when calling system functions **sendMessage** and **postMessage** right Component execute.

### 7.4. Class SimulationController

#### 7.4.1. initialize

Initialize the Scenario.

#### 7.4.2. terminate

Terminate the Scenario.

#### 7.4.3. execute

This method will call your Scenario's methods **preProcess**, **mainProcess** and **postProcess**.

#### 7.4.4. postMessage

Asynchronously posts a message (This method returns control without blocking). The class `NetworkMessageEnvelope` will be used to envelop the message. This method is called by the system function with the same name.

#### 7.4.5. sendMessage

Synchronously sends a message (This method blocks control until the destination component processes the message). The class `NetworkMessageEnvelope` will be used to envelop the message. This method is called by the system function with the same name.

#### 7.4.6. processMessage

Processes a received message. Call your Scenario's method **eventProcess** when a message is received. The class `NetworkMessageEnvelope` will be used to envelop the message. This method is called by the system function with the same name.

# How to use DynaBioS-2-Java-platform

## **7.5. Class ModelManager**

### **7.5.1. initialize**

Initialize the Model.

### **7.5.2. terminate**

Terminate the Model.

### **7.5.3. getModel**

Return the Model. This method is called by the system function with the same name.

## **7.6. Class ComponentHandler**

### **7.6.1. getComponentID**

Return the component handlers component id.

# How to use DynaBioS-2-Java-platform

## 8. Example code

This chapter shows two examples of code using DynaBioS-2-Java-platform. The first example is for a controller with an internal component handler and the second example does the same thing, but by using a remote component handler instead. If you need more information on what the used method does, see previous section.

### 8.1. Internal component handler

This chapter shows an example where DynaBioS-2-Java-platform is used with one component, HelloWorldComponent, running the component internal.

#### 8.1.1. Class HelloWorld

// Example of a Main class using an internal component handler.

```
public class HelloWorld {  
    public static final String CMPTYPE_HELLOWORLD = "HelloWorldComponent_A";  
  
    public static void main(String[] args) {  
        final HelloWorldScenario scenario = new HelloWorldScenario();  
        final HelloWorldModel model = new HelloWorldModel("Hello World!");  
        final ComponentEntry[] entries = new ComponentEntry[] {  
            // Use internal component handler for class HelloWorldComponent  
            new ComponentEntry(  
                CMPTYPE_HELLOWORLD,  
                ComponentEntry.HANDLER_TYPE_INTERNAL,  
                128,  
                HelloWorldComponent.class.getName(),  
                null)  
            };  
        // Load the system  
        SystemLoader.load(scenario, model, entries);  
    }  
}
```

# How to use DynaBioS-2-Java-platform

## 8.1.2. Class HelloWorldModel

```
// Example of a Model class
public class HelloWorldModel extends SimulationModel {
    private String string;

    public HelloWorldModel(final String modelString) {
        string = modelString;
    }

    public void initialize() {
        System.err.println("The HelloWorldModel is initialized.");
    }

    public void terminate() {
        System.err.println("The HelloWorldModel is terminated.");
    }

    public String getString() {
        return string;
    }
}
```

## 8.1.3. Class HelloWorldScenario

```
// Example of a Scenario class
public class HelloWorldScenario extends SimulationScenario {
    private HelloWorldModel model;
    private ComponentHandler helloWorldComponentHandler;

    public void initialize() {
        System.err.println("The HelloWorldScenario is initialized.");
    }

    public void terminate() {
        System.err.println("The HelloWorldScenario is terminated.");
    }
}
```

## How to use DynaBioS-2-Java-platform

```
public void preProcess() {  
    // Aquire a model instance  
    model = (HelloWorldModel) getModel();  
    // Allocate a model instance  
    helloWorldComponentHandler = allocateComponent(HelloWorld.CMPTYPE_HELLOWORLD);  
}  
  
public void mainProcess() {  
    // Make a message  
    final Message message = new Message();  
    HelloWorldComponentMessage.makeOutputStringMessage(  
        message, helloWorldComponentHandler, model.getString());  
  
    // Send a message  
    sendMessage(message);  
  
    // Process a received message. The system will wait until a message is received, then eventProcess will be  
    // called.  
    processMessage();  
}  
  
public int eventProcess(Message message) {  
    // Check if the received message component id matches the allocated component handlers component id  
    if (message.getComponentID() == helloWorldComponentHandler.getComponentID()) {  
        // Check if the received message command is a reply output string type, in that case print out the  
        // message.  
        switch (message.getCommand()) {  
            case HelloWorldComponentMessage.CMDTYPE_REP_OUTPUT_STRING:  
                System.err.println("A REP_OUTPUT_STRING message is received.");  
                System.err.println(message.getData().getString());  
                return 0;  
            default:  
                return 1;  
        }  
    } else {
```

## How to use DynaBioS-2-Java-platform

```
        return 2;
    }
}

public void postProcess() {
    // Release the component handler
    releaseComponent(helloWorldComponentHandler);
}
}
```

### 8.1.4. Class HelloWorldComponent

// Example of a Component class

```
public class HelloWorldComponent extends Component {

    protected void initialize() {
        System.err.println("A HelloWorldComponent is initialized.");
    }

    protected void terminate() {
        System.err.println("A HelloWorldComponent is terminated.");
    }

    protected int processMessage(final Message message) {
        // Check if the received message command is a output string type, in that case send a reply message.
        switch (message.getCommand()) {
            case HelloWorldComponentMessage.CMDTYPE_OUTPUT_STRING:
                return processOutputStringMessage(message);
            default:
                return 1;
        }
    }

    private int processOutputStringMessage(final Message message) {
        final String string = HelloWorldComponentMessage.parseOutputStringMessage(message);
        System.out.println(string);
    }
}
```

## How to use DynaBioS-2-Java-platform

```
        System.out.flush();
        postMessage>HelloWorldComponentMessage.CMDTYPE_REP_OUTPUT_STRING, null);
        return 0;
    }
}
```

### 8.1.5. Class HelloWorldComponentMessage

```
// Example of a ComponentMessage class
public class HelloWorldComponentMessage {
    public static final int CMDTYPE_OUTPUT_STRING = 1;
    public static final int CMDTYPE_REP_OUTPUT_STRING = 1;

    public static void makeOutputStringMessage(final Message message,
        final ComponentHandler handler, final String string) {
        final MessageData data = new MessageData(string.length());
        data.putString(string);
        Message.makeMessage(message, handler, CMDTYPE_OUTPUT_STRING, data);
    }

    public static String parseOutputStringMessage(final Message message) {
        final MessageData data = message.getData();
        return data.getString();
    }
}
```

### 8.2. Remote component handler

The only thing the user needs to change to get the example in the previous chapter to run the component remote instead of local is the variables for ComponentEntry. The remote component server need a Main class that will call the class RemoteComponentServer, here named HelloWorldRemoteServer. Of course access to class HelloWorldComponent and HelloWorldComponentMessage is also required.

This chapter shows the same example as chapter 8.1, but here the component will be run remote instead, not listed classes remain the same

# How to use DynaBioS-2-Java-platform

## 8.2.1. Class HelloWorld

// Example of a Main class using a remote component handler.

```
public class HelloWorld {  
    public static final String CMPTYPE_HELLOWORLD = "HelloWorldComponent_A";  
  
    public static void main(String[] args) {  
        final HelloWorldScenario scenario = new HelloWorldScenario();  
        final HelloWorldModel model = new HelloWorldModel("Hello World!");  
        final ComponentEntry[] entries = new ComponentEntry[] {  
            // Use remote component handler. Remote host is set to 192.168.0.100 on port number 1234.  
            new ComponentEntry(  
                CMPTYPE_HELLOWORLD,  
                ComponentEntry.HANDLER_TYPE_REMOTE,  
                128,  
                null,  
                new String[] { 192.168.0.100, "1234" }  
            );  
        };  
        // Load the system  
        SystemLoader.load(scenario, model, entries);  
    }  
}
```

## 8.2.2. Class HelloWorldRemoteServer

// Example of a Main class for the remote component server.

```
public class HelloWorldRemoteServer {  
    public static void main(String[] args) {  
        // Run a remote component server for class HelloWorldComponent that will bind a connection from a  
        // controller on 192.168.0.50 on port number 9876.  
        RemoteComponentServer.main(new String[] { HelloWorldComponent.class.getName(), 192.168.0.50, 9876 });  
    }  
}
```