Development of Robot-enhanced Therapy for
Children with Autism Spectrum Disorders

Project No. 611391

DREAM
Development of Robot-enhanced Therapy for Children with Autism Spectrum Disorders

Grant Agreement Type:	Collaborative Project
Grant Agreement Number:	611391

D3.3 Quality Assurance Procedures

Due date: **1/10/2014**
Submission Date: **24/09/2014**

Start date of project: **01/04/2014** Duration: **54 months**

Organisation name of lead contractor for this deliverable: **University of Skövde**

Responsible Person: **D. Vernon** Revision: **2.0**

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including the Commission Service)	
RE	Restricted to a group specified by the consortium (including the Commission Service)	
CO	Confidential, only for members of the consortium (including the Commission Service)	

Contents

Executive Summary

Deliverable D3.3 sets out the procedures used in DREAM to validate and test software developed by the partners prior to integration into the release version of the DREAM software repository. It is, in essence, is a check-list of the *mandatory* standards set out in Deliverable D3.2 and software must satisfy all these checks before it can be integrated.

Developers can submit their software by transferring it to a sub-directory in the `submitted` branch of the subversion repository (see Figure 1 below) and by sending an email to `integration@dream2020.eu` with the path to the component being submitted in the subject line.

The system integration team at HIS will validate the software in the relevant sub-directory against the check-list in this deliverable, compiling the code and running the unit test application supplied with the submission. If the component satisfies all the checks and the test runs successfully, the component will be moved to the release directory and it can be then used in DREAM applications.

The complete DREAM software system will also be subject to quality assurance procedures, as described in Deliverable D3.3, including white-box structural tests, regression tests, and acceptance tests.

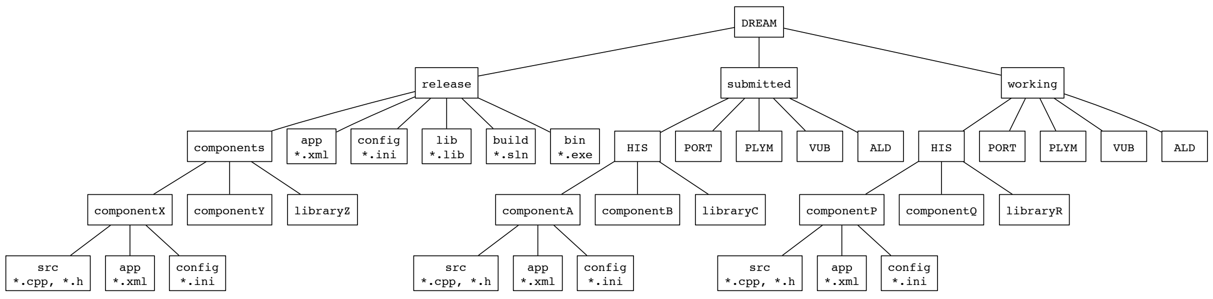

Figure 1: Partial directory structure for the DREAM software repository.

Principal Contributors

The main authors of this deliverable are as follows (in alphabetical order).

Erik Billing, University of Skövde
Aran Smith, University of Skövde
David Vernon, University of Skövde

Revision History

Version 1.0 (DV 25-08-2014)
First draft.

Version 1.1 (DV 23-09-2014)
Changed directory structure in Figure 1 to include a `build` directory. Removed empty references section.

Version 2.0 (DV 24-09-2014)
Implementation of several suggestions arising from pre-submission review by the DREAM team.

1 Files and Directories

Refer to Deliverable D3.2, Appendix A (Mandatory Standards for File Organization), for a definition of the standards on which this checklist is based.

- ☐ Files for a single component are stored in a directory named after the component with the leading letter in lowercase: `<componentName>`.
- ☐ This directory has three sub-directories: `src`, `app`, and `config`.
- ☐ The `src` directory contains one header file and three source files, named as follows.
  - ☐ `<componentName>.h`
  - ☐ `<componentName>Main.cpp`
  - ☐ `<componentName>Configuration.cpp`
  - ☐ `<componentName>Computation.cpp`
- ☐ The `app` directory contains an XML application file named after the component but with the suffix `TEST`: `<componentName>TEST.xml`.
- ☐ The `app` directory contains a `README.txt` file with instructions on how to run the test .
- ☐ The `config` directory contains a `<componentName>.ini` configuration file.
- ☐ The configuration file contains the key-value pairs that set the component parameters.
- ☐ Each key-value pair is written on a separate line.

2 Internal Source Code Documentation

Refer to Deliverable D3.2, Appendix B (Mandatory Standards for Internal Source Code Documentation), for a definition of the standards on which this checklist is based.

The `<componentName>.h` file contains a documentation comment with the following sections:

- ☐ `/**`
  ` * @file <componentName>.h`
- ☐ ` * \section lib_sec Libraries`
- ☐ ` * \section parameters_sec Parameters`
- ☐ ` * <b>Command-line Parameters </b>`
- ☐ ` * <b>Configuration File Parameters </b>`
- ☐ ` * \section portsa_sec Ports Accessed`
- ☐ ` * \section portsc_sec Ports Created`
- ☐ ` * <b>Input ports</b>`
- ☐ ` * <b>Output ports</b>`
- ☐ ` * <b>Port types </b>`
- ☐ ` * \section in_files_sec Input Data Files`
- ☐ ` * \section out_data_sec Output Data Files`
- ☐ ` * \section conf_file_sec Configuration Files`
- ☐ ` * \section example_sec Example Instantiation of the Component`
- ☐ ` * \author`
  ` * <forename> <surname>`

All source files contain a block comment that gives the copyright notice, as follows.

```
/*
 * Copyright (C) 2014 DREAM Consortium
 * FP7 Project 611391 co-funded by the European Commission
 *
 * Author:  <name of author>, <author institute>
 * Email:   <preferred email address>
 * Website: www.dream20202.eu
 *
 * This program comes with ABSOLUTELY NO WARRANTY.
 */
```

- ☐ `<componentName>.h`
- ☐ `<componentName>Main.cpp`
- ☐ `<componentName>Configuration.cpp`
- ☐ `<componentName>Computation.cpp`

3 Component Functionality

Refer to Deliverable D3.2, Appendix C (Mandatory Standards for Component Functionality), for a definition of the standards on which this checklist is based.

☐ `<componentName>.h` contains a declaration of a class derived from `yarp::os::RFModule`:

```
class <ComponentName> : public RFModule {} // first char in uppercase
```

☐ `<componentName>.h` contains a declaration of a class derived from either `yarp::os::Thread` or `yarp::os::RateThread`:

```
class <ComponentName>Thread : public Thread {} // first char in uppercase
```

☐ `<componentName>Configuration.cpp` contains a definition of the methods for `class <ComponentName> : public RFModule {}`

☐ `<componentName>Computation.cpp` contains a definition of the methods for `class <ComponentName>Thread : public Thread {}` or `class <ComponentName>RateThread : public Thread {}`

☐ The `<ComponentName>` class is instantiated as an object in `<ComponentName>Main.cpp`:

```
<ComponentName> <componentName>;
```

☐ The `<ComponentName>Thread` / `RateThread` class is instantiated as an object in the `<ComponentName>::configure()` method in `<componentName>Configuration.cpp`:

```
<componentName>Thread = new <ComponentName>Thread();
<componentName>Thread->start();
```

☐ A `ResourceFinder` class, e.g. `ResourceFinder rf`, is instantiated in `<componentName>Main.cpp`

☐ The component sets the default configuration filename, named after the component with a `.ini` extension, in `<componentName>Main.cpp`:

```
rf.setDefaultConfigFile("<componentName>.ini");
```

☐ The component sets the default path (context) in `<componentName>Main.cpp`:

```
rf.setDefaultContext("components/<componentName>/config");
```

☐ The component reads all its key-value parameters from either a `<componentName>.ini` configuration file or from the list of command line arguments using the `ResourceFinder check()` method, called from within the `configure()` method in `<componentName>Configuration.cpp`:

```
<parameterValue> = rf.check("<key>",                 // parameter key
                            Value(<number>),          // default value
                           "Key value (int)").asInt(); // key value type
```

- ☐ The component reads all its key-value parameters from either a `<componentName>.ini` configuration file or from the list of command line arguments using the `ResourceFinder check()` method, called from within the `configure()` method in `<componentName>Configuration.cpp`:

```
<parameterValue> = rf.check("<key>",                // parameter key
                         Value(<number>),           // default value
                        "Key value (int)").asInt(); // key value type
```

- ☐ The component allows the port names to be set and overridden using the port name key-value parameters in the `<componentName>.ini` configuration file.
- ☐ All port names have a leading `/`.
- ☐ Input and output port names have a trailing `:i` or `:o`, respectively.
- ☐ The component allows the default name of the component to be set and overridden with the `--name` parameter:

```
 moduleName              = rf.check("name",
                           Value("<componentName>"),
                           "module name (string)").asString();

setName(moduleName.c_str());
```

- ☐ The component allows commands to be issued on a special port with the same name as the component by overloading the `respond()` method in the resource finder `RFModule` class in `<componentName>Configuration.cpp`.
- ☐ The component effects all communication with other component using YARP ports.

4 Component Unit Testing

- ☐ A unit test application named `<componentName>TEST.xml` is provided in the `app` directory.
- ☐ The test application launches the component being tested on a YARP run servers called `dream1` using the `<node> </node>` construct.
- ☐ The test application connects the component through its ports to a data source and a data sink (linked either to files or driver/stub components).
- ☐ If required, the data source and sink file resources are provided in the `config` directory.
- ☐ If required, the driver and stub components are located in `src` directory.
- ☐ Unit test instructions are provided in a file named `README.txt` in the `app` directory.
  - ☐ The instructions explain how the communication and computation functionality are validated by describing the (sink) output data that will be produced from the (source) input data.
  - ☐ The instructions explain how the configuration functionality is validated by describing what changes in behaviour will occur if the values for the component parameters in the component configuration (`.ini`) file are altered.
  - ☐ The instructions explain how the coordination functionality is validated by describing what changes in behaviour will occur when commands are issued interactively by the user to the component using the port named after the component itself.

5 System Testing

White-box testing will be performed on a system-level by removing the driver and stub functions that simulate the output and input of data in the top-level system architecture, allowing that source and sink functionality to be provided instead by the component being integrated. This will establish whether or not the component in question adheres to the required data-flow protocol.

Regression testing refers to the practice of re-running all integration tests — black-box and white-box — periodically to ensure that no unintentional changes has been introduced during the ongoing development of the DREAM software release. These test check for backward compatibility, ensuring that what used to work in the past remains working. Regression tests will be carried out on all software in the DREAM release every two months.

The DREAM software will be subject periodic qualitative assessment by the DREAM psychotherapist practitioners. The specific goal of these tests will be to validate the behaviour and performance of the system against the user requirements set out in deliverables D1.1, D1.2, and D1.3. These tests will take place whenever a new version of the DREAM software release it made available to the practitioners.