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

D3.2 Software Engineering Standards

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 5

Principal Contributors 6

Revision History 6

I Guiding Principles 7

1 Objectives 7

2 Component-based Software Engineering 7
2.1 Background Material . . . 7
2.2 Characteristics of Component-Based Software Engineering . . . 7
2.3 Component Granularity and Systems . . . 8
2.4 Component Interfaces . . . 9

3 The BRICS Component Model 9

4 Model-Driven Engineering and the Component-Port-Connector Model 10

5 Guidelines 12

II Software Development Environment 13

1 Programming languages and compilers 13

2 Operating Systems 13

3 Robot Programming Framework 13
3.1 Support for the Component-Port-Connector Meta-Model . . . 14
3.2 Robot Applications . . . 15
3.3 Coarse-grained Functionality . . . 15
3.4 Multiple Instances of Components . . . 15
3.5 External Configuration . . . 16
3.6 Different Contexts . . . 17
3.7 Runtime Configuration . . . 18
3.8 Assignment of Resources . . . 19
3.9 Asynchronous Communication . . . 19
3.10 Distributed Computing . . . 20
3.11 Stable Interfaces . . . 20
3.12 Abstract Component Interfaces . . . 20
3.13 Multiple Transport Layer Protocols . . . 21
3.14 Graphic User Interface Tools . . . 21

4 Make File Utilities 22

5 Software Libraries **22**

6 Software Repository **22**

III Standards **23**

1 Component and Sub-system Specification **23**

2 Component Design **24**

3 Component Implementation **24**

4 Testing **24**
4.1 Black-box Unit Tests . . . 25
4.2 White-box Structural Tests . . . 26
4.3 Regression Tests . . . 26
4.4 Acceptance Tests . . . 26

5 Documentation **26**

References **28**

Appendix A Mandatory Standards for File Organization **30**
A.1 Directory Structure . . . 30
A.2 Filename Roots and Extensions . . . 30
A.3 File Organization . . . 30
A.3.1 Source Files . . . 30
A.3.2 Application Files . . . 31
A.3.3 Configuration Files . . . 31

Appendix B Mandatory Standards for Internal Source Code Documentation **32**
B.1 General Guidelines . . . 32
B.2 Documentation Comments . . . 32
B.3 Implementation Comments . . . 34

Appendix C Mandatory Standards for Component Functionality **37**
C.1 Component Configuration Standards . . . 38
C.1.1 Default Configuration File . . . 38
C.1.2 Default Configuration Context . . . 39
C.1.3 User-defined Configuration File . . . 40
C.1.4 User-defined Context . . . 40
C.1.5 Component Parameters . . . 40
C.1.6 Component Port Names . . . 41
C.1.7 Component Name . . . 41
C.2 Component Coordination Standards . . . 42
C.3 Component Computation Standards . . . 43
C.4 Component Communication Standards . . . 43

Appendix D Recommended Standards for Programming Style **44**
D.1 Indentation and Line Breaks 44
D.2 Declarations 44
D.3 Placement 45
D.4 Statements 45
D.5 Naming Conventions 47
D.6 And Finally: Where To Put The Opening Brace { 48

Appendix E Recommended Standards for Programming Practice **50**
E.1 C++ Language Conventions 50
E.2 C Language Conventions 50
E.3 General Issues 50

Executive Summary

Deliverable D3.2 is a reference manual of software engineering standards. It is a working document in the sense that it is intended to be consulted by all software developers throughout the project. It comprises three parts.

Part I sets out the general principles of component-based software engineering that guide the decisions in Parts II and III.

Part II focusses on the software development environment that supports the development process: the supported languages, operating systems, libraries, robot programming framework, and related tools and utilities.

Part III then proceeds to address the standards associated with each phase of the software development life-cycle, from component and sub-system specification, through component design, component implementation, and component and sub-system testing, to documentation. Particular emphasis is placed on the latter phases of the life-cycle — implementation, test, and documentation — because these are particularly important for effective system integration and long-term support by third-party software engineers and system users.

The inclusion of Part II was not envisaged in the Description of Work and this material is the result of work done in Task 3.1 (Architectural Design). The deliverable associated with Task 3.1 — D3.1 (System Architecture) — deals with the design of the DREAM system architecture, hence the decision to document the development environment here. On the other hand, the standards set out in Part III are the outcome of work in Task 3.2 (Software Engineering Standards and Quality Assurance Procedures).

Note that Task 3.2 has a second deliverable — D3.3 (Quality Assurance Procedures) — which focusses on the procedures by which software developed in DREAM is submitted for integration, tested, and checked against the standards set out in Part III of this document.

In general, the standards defined here attempt to find a balance between specifying too much (with the result that no one will use them) and specifying too little (with the result that the desired software quality won't be achieved and software integration will be hindered). The standards should be simple enough to be easily adopted and applied, but comprehensive enough to be useful in the creation of high-quality easily-maintained software. Furthermore, in this deliverable as in others, we have opted to include only the essential information, targetting brevity rather than extensive discussion.

Principal Contributors

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

Revision History

Version 1.0 (DV 20-08-2014)
First draft.

Version 1.1 (DV 26-08-2014)
Amended file structure and fixed various typos.

Version 1.2 (DV 27-08-2014)
Fixed error in section on creating multiple instances of the same component.

Version 1.3 (DV 30-08-2014)
Fixed error in file standards and added material on the DREAM wiki to Section 5: Documentation.

Version 1.4 (DV 23-09-2014)
Changed directory structure in Figure 5 to include a `build` directory. Reordered sections so that the references appear before the appendices.

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

Part I
Guiding Principles

1 Objectives

The purpose of this deliverable is to define a set of project standards governing the specification, design, documentation, and testing of all software to be developed in work packages WP4, WP5, and WP6. Specification standards address functional definition, data representation, and component & sub-system behaviour. Design standards focus on the decoupling of functional computation, component communication, external component configuration, and inter-component coordination [1]. Software test strategies will include black-box unit testing, white-box structural testing at the sub-system level, and regression testing for to ensure backward compatibility. Acceptance tests will be carried out on the basis of the required behaviours determined by ASD practitioners in work package WP2.

To facilitate efficient implementation and reuse of robot software by application developers, the DREAM project has decided to adopt the principles of a best-practice component-based software engineering model, such as the BRICS Component Model (BCM) [2, 3, 4, 5]. The goal of Part I is to identify the principles of component-based software engineering and provide a set of guidelines that can then be used in setting the standards and defining the practices to be used by all software developers in the project, i.e. the standards set out in Part III.

The principles of component-based software engineering have another important role in the project in that they also guide the selection of a software environment that will be used to provide the abstraction layer between the application code to be developed in WP4, WP5, and WP6, on the one hand, and the middleware and operating system services (including support of distributed systems processing and device input/output), on the other. This abstraction layer is typically provided by the robot programming framework, or robot platform, for short. It provides an abstract interface between the robot software system (implemented as a network of components) and the underlying operating system and middleware; see Fig. 1. This robot platform is identified in Part II, Section 3.

2 Component-based Software Engineering

2.1 Background Material

The following overview of component-based software engineering focusses on the issues that are particularly important in robot programming. It is based in part on the tutorials by Davide Brugali, Patrizia Scandura, and Azamat Shakhimardanov “Component-Based Robotic Engineering” [7, 8], complemented by material on the BRICS Component Model by Herman Bruyninckx and colleagues [5], and the paper by Christian Schlegel and colleagues on model-driven software development in robotics [6], among other sources.

2.2 Characteristics of Component-Based Software Engineering

Targetting the development of reusable software, component-based sofware engineering (CBSE) complements traditional object-oriented programming by focussing on run-time composition of software rather than link-time composition. Consequently, it allows different programming languages, operating systems, and possibly communication middleware to be used in a given application. It harks back

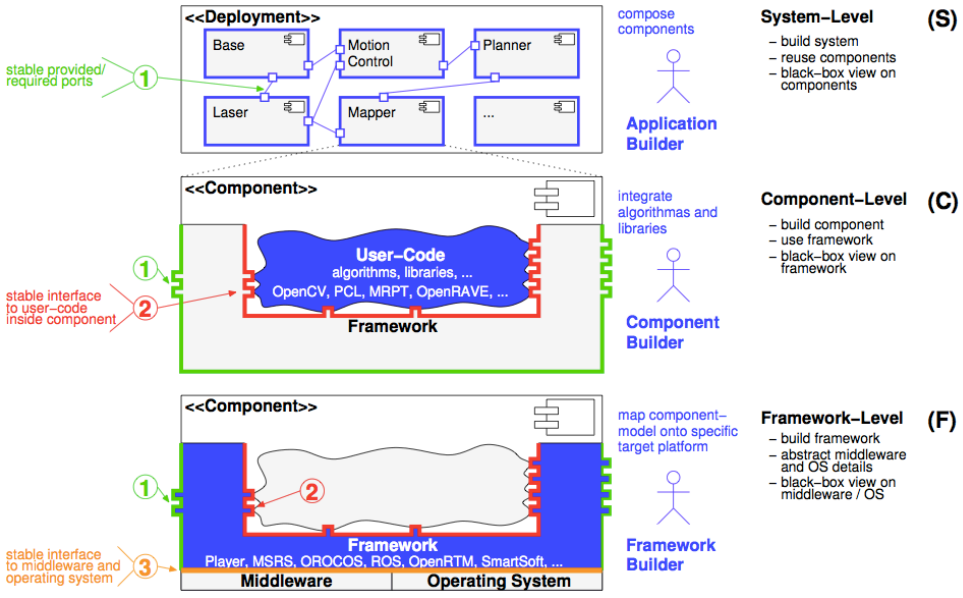

Figure 1: The relationship between the robot software system (implemented as a network of components), the robot programming framework (or platform), and the underlying middleware and operating system(s) (from [6]). The existence of stable interfaces is a key attribute of each level.

to the classic concept of communicating sequential processes (CSP) [9], the idea being that components are individually-instantiated processes that communicate with each other by message-passing. Typically, component models assume asynchronous message-passing where the communicating component is non-blocking, whereas CSP assumed synchronous communication. The key idea is that components can act as reusable building blocks and that applications, or system architectures, can be designed by *composing* components.

This gives rise to the two key concerns of component-based models: *composability* and *compositionality*. In complex robotics system, integration is difficult. It is made easier by adopting practices that "make components more composable and systems more compositional" [4].

Thus, *composability* is the property of a component to be easily integrated into a larger system, i.e. to be reused under composition, while *compositionality* is the property of a system to exhibit predictable performance and behaviour if the performance and behaviour of the components are known.

Unfortunately, it is not possible to maximize both properties simultaneously because the information hiding characteristic inherent in good component design conflicts with the need for system engineers to optimize system robustness by selecting components with the most resilient and flexible internal design [4].

2.3 Component Granularity and Systems

The granularity of *components* is larger than that of *objects* in object-oriented approaches. Thus, the functionality encapsulated in a component is typically greater than that of an object. Also, components are explicitly intended to be stand-alone reusable pieces of software with well-defined public interfaces.

In general, a component implements a well-encapulated element of robot functionality. Systems

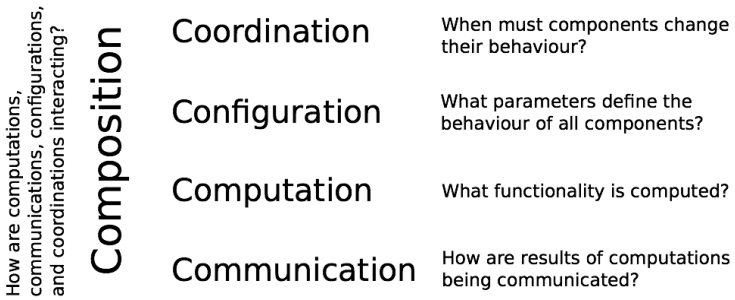

Figure 2: The 5 Cs: the core idea is that good component-based engineering practice strives to decouple the design concerns of these five functions (from [5]).

are constructed by connecting components. The connections are made using ports in the components. As we will see below, this gives rise to the so-called *Component-Port-Connector* model.

2.4 Component Interfaces

In component-based sofware engineering, as in object-oriented programming, the component specification is separated from the component implementation. A component exposes its functionality through abstract interfaces that hide the underlying implementation. Thus, the implementation can be altered without affecting any of the systems or other components that use (i.e. interface with) that component. Components exchange data through their interface.

Interfaces can be classified as either *data interfaces* or *service interfaces*. Data interfaces expose state information about the component and typically provide get / set operations for retrieving or setting the values of attributes. These attributes will typically be specified as abstract properties (in order to keep the implementation hidden). A service interface is a declaration of the set of functionalities offered by a component on the parameters that are passed to it.

An interface can be either stateful or stateless. In a stateful interface, the invocation of an operation changes the component's internal state and the information returned by the operation is computed differently, depending on the component's state. Thus the behaviour of the exposed operations depends on the history of their previous invocations. In a stateless interface, the operations's behaviour is always the same and the outcome depends only of the information provided through the parameters, i.e. the data exchanged through the interface. In a stateless interface, a client component has to specify all the information related to its request for the invocation of some operation. Consequently, a component with a stateless interface can interact with many different clients and client requests since none of them can make any assumptions about the state of the component.

3 The BRICS Component Model

The BRICS Component Model (BCM) provides guidelines, meta-models, and tools for structuring the development of individual components and component-based systems in a framework-nonspecific manner, i.e. at an abstract (meta) level that doesn't refer explicitly to the implementation of the required functionality on a specific robot platform.

BCM brings together two strands of software engineering: the separation of the so-called 4Cs of communication, computation, configuration and coordination [1] and Model-Driven Engineering.

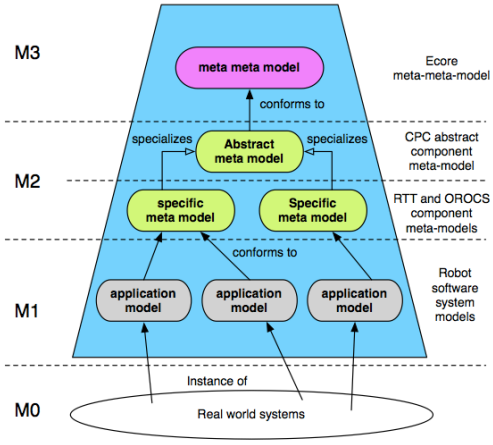

Figure 3: The four levels of abstraction from the OMG [10] applied in the BRICS Component Model (from [5]).

BCM extends the 4Cs to 5Cs by splitting configuration into finer-grained concepts of configuration and composition: see Fig. 2. The core idea is that good component-based engineering practice strives to decouple the design and implementation of these five concerns. We will summarize them briefly.

Computation refers to the system's main functionality, i.e. the information processing the component has been designed to perform. *Communication* provides the data required by the computation and takes care of the quality of service of data communication, e.g. timing, bandwith, loss (accuracy), priority, and latency. *Coordination* refers to the functionality that determines how all the components in a system work together and, thus, how the component or system behaves. *Configuration* refers to the functional aspects concerned with influencing the behaviour of the computation and communication elements either at start-up or at run-time. Finally, *composition*, the fifth C introduced by Herman Bruyninckx, provides the glue that binds together the other four Cs, each of which is focussed on decoupling functionalty. In contrast, composition *is* concerned with coupling: how the other four Cs interact. Strictly, the C does not reflect software functionality but is rather a design characteristic that seeks to find a good tradeoff between composability and compositionality.

4 Model-Driven Engineering and the Component-Port-Connector Model

Model-Driven Engineering (MDE) aims to improve the process of code generation from abstract models that describe a domain. The Object Management Group (OMG) [10] defines four levels of model abstraction, going from higher to lower levels of domain specificity, i.e. from platform-independent to platform-specific, by adding platform knowledge. These four levels can be characterized as follows [5] (see also Fig. 3).

M3 Domain-nonspecific: the highest level of abstraction using a meta-meta-model.

M2 Platform-independent representation of a domain: uses a *Component-Port-Connector* (CPC) meta-model.

M1 Platform-specific model: a concrete model of a specific robotic system but without using a specific programming language.

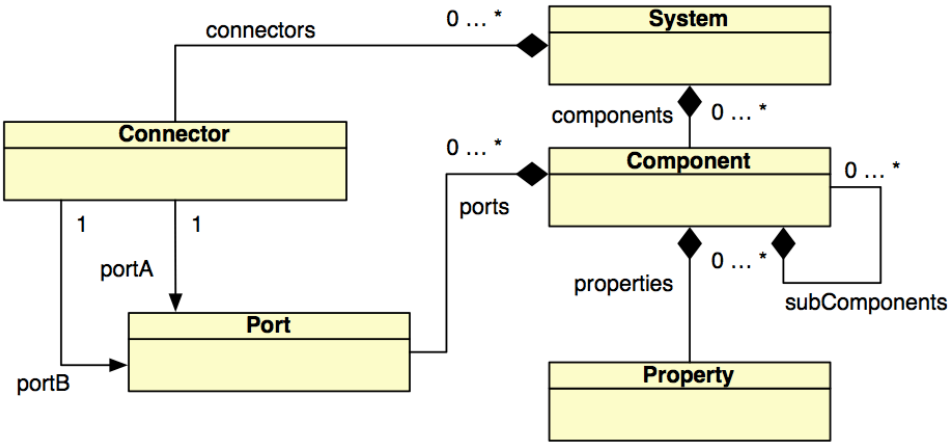

Figure 4: A UML diagram representing the Component-Port-Connector meta-model (from [5]).

M0 The implementation level of a specific robotic system, with defined software frameworks, libraries, and programming languages.

Of particular interest here is level M2 because the abstract model, i.e. the *Component-Port-Connector* (CPC) meta-model, maps directly to the concepts of component-based software engineering. Figure 4 shows a UML diagram representing the CPC meta-model. The essence of this model is as follows.

- An application system has zero or more components and zero or more connectors.
- A component has zero or more ports.
- A port belongs to one and only one component.
- A connector is alway between two ports.
- Components expose their data and service interfaces on their ports and exchange data over the attached connectors.

We are now in a position to identify the guidelines for selecting an appropriate robot programming framework and setting standards for the development component-based software centred on the Component-Port-Connector meta model and model-driven engineering.

5 Guidelines

The principles of CBSE and Level 2 of MDE can be encapsulated in the following guidelines. These guidelines provide the basis for the choice of robot programming framework and the software development standards in Parts II and III.

1. The software engingeering methodology, in general, and the robot programming framework, in particular, should support the **component-port-connector** meta-model, the principles of component-based software engineering, and the principles of model-driven engineering.
2. The **robot application** should be implemented by identifying the components to be instantiated and specifying the connections between their port interfaces.
3. Each component should encapsulate some **coarse-grained** robot functionality (i.e. the component should perform some substanial function while maintaining its cohesion and not doing too many disparate things).
4. It should be possible to have **multiple instances** of the same component (this requires a mechanism for uniquely naming each instance of the component and propagating that name to the component's port names).
5. There should be a facility for **external configuration** of the behaviour of the component (this allows the behaviour of individual instances of a component to be customized by the application developer without recourse to the component developer).
6. There should be a facility to allow components to run in **different contexts** (in essence, this means that the robot programming framework should allow the component user to specify the location of the components configuration files and other resources).
7. There should be a facility for **runtime configuration** of the behaviour of the component (this allows users and other components to control the component's processing, if necessary).
8. Configuration should provide explicit **assignment of resources** to the component, where necessary.
9. Components should be able to run and communicate **asynchronously.**
10. It must be possible to run components on a **distributed system** (this requires a mechanism to assign a component to a given compute-server).
11. Component should have **stable interfaces** (this implies, among other things, that backward compatibility should be assured).
12. The robot programming framework should provide facilities to support **abstract component interfaces** (in effect, this will mean that there is support for high-level communication using ports).
13. There should be flexible communication infrastructure with **multiple transport layer protocols** (this means in effect that the ports can be configured to use different communication protocols, e.g. udp, tcp, and mcast).
14. The robot programming framework should provide **graphic user interface tools** for managing the execution of robot applications.

Part II
Software Development Environment

In this part, we will identify the six constituents of the DREAM software development environment:

1. Programming languages and compilers;
2. Operating systems;
3. Robot programming framework;
4. Make file utilities;
5. Software libraries
6. Software repository.

The critical choice of robot programming framework is based directly on the fourteen guidelines set out in the previous section.

A complete set of instructions on how to download and install all the various utilities, libraries, and tools comprising the DREAM software development environment will be provided on the DREAM wiki in due course. Furthermore, it will provide instructions on how to compile and link a prototype component and to configure, manage, and run a prototype robot application comprising a collection of connected components in the DREAM software respository.

1 Programming languages and compilers

DREAM software should be written in either the C language or the C++ language and compiled using Microsoft Visual C++ Express compiler. Either version 9.0 (also known as Visual C++ 2008) or version 10.0 (also known as Visual C++ 2010) should be used. The version you choose is important because you may have to download binaries that are compatible with that version when installing a binary image of YARP (see Section 3 below). You can download the required compiler from here.

Version 9: `http://go.microsoft.com/?linkid=7729279`
Version 10: `http://www.visualstudio.com/downloads/download-visual-studio-vs`

2 Operating Systems

Software will be developed for Windows 7 only because this is the operating system used by the therapists.

3 Robot Programming Framework

There are many robot programming frameworks. These include ROS [11], YARP [12], URBI [13], and Orca [14, 15]. All have their respective advantages and disadvantages, as explained in various surveys and comparisons [16, 17].

Our goal is to select one that will allow the software required to deliver the DREAM functionality to be developed efficiently and effectively. As we have already stated, we have adopted the principles of component-based software engineering (CBSE) and several guidelines on the required attributes of a good CBSE framework have been identified in Part I.

Having considered the various options, we decided to adopt YARP [12] as the framework for DREAM. Note that ROS, a popular choice in many projects, has been criticized by others in the context of Component Based Software Engineering as lacking an abstract component model [6]. On the other hand, hands-on experience of YARP in the DREAM consortium suggested that YARP does provide a framework that follows all the CBSE guidelines identified above. To demonstrate this, the following sections provide examples of the manner in which it does so.

First however, note that instructions for installing YARP can be found here `http://wiki.icub.org/yarpdoc/install.html`. Note that these instructions also deal with installing various other dependencies such as CMake and the ACE library.

In the following, we use the terms *component* and *module* interchangeably. In general, we use *component* when referring to the external abstract CBSE view of the entity and *module* when referring to the YARP implementation of a component.

3.1 Support for the Component-Port-Connector Meta-Model

YARP supports the component-port-connector meta-model as well as the principles of component-based software engineering and the principles of model-driven engineering. Specifically, a YARP module corresponds to a component and it can have an arbitrary number of ports, with modules being connected by specifying the ports on either side of a connection. The following XML application code segment demonstrates this from the perspective of an application developer. As we will see in the following sections, YARP provides considerable support for component development.

```
<module>
   <name>protoComponent</name>
   <node>dream1</node>
   <tag>protoComponent</tag>
</module>

<module>
   <name>yarpview</name>
   <parameters>--name /binaryImage </parameters>
   <node>dream1</node>
   <tag>output_image</tag>
</module>

<connection>
  <from>/robot/camera/left</from>
  <to>/protoComponent/image:i</to>
  <protocol>tcp</protocol>
</connection>

<connection>
  <from>/protoComponent/image:o</from>
  <to>/binaryImage</to>
  <protocol>tcp</protocol>
</connection>
```

Here we have two components (YARP modules): `protoComponent` and `yarpview` with connections from port `/robot/camera/left` (made available by some other application) to `/protoComponent/image:i` and from `/protoComponent/image:o` to `/binaryImage`.

3.2 Robot Applications

YARP supports the implementation of robot applications by identifying the components to be instantiated and specifying the connections between their port interfaces. This is evident from the previous example.

3.3 Coarse-grained Functionality

YARP support the encapsulation of coarse-grained robot functionality insofar as the scope of the functionality in a YARP module is under the control of the component developer.

3.4 Multiple Instances of Components

YARP supports the instantiation of **multiple instances** of the same component. This is achieved using an XML `<tag>...</tag>` construct to uniquely name each instance of the component. To allow the component name to be propagated to the component's port names to keep them unique, we use the `--name <value>` in the `<parameters> ... </parameters>` construct. You can see the same construct being used to specify the input port on the `yarpview` module.

```
<module>
   <name>protoComponent</name>
   <parameters>--name /protoComponentA</parameters>
   <node>dream1</node>
   <tag>protoComponentA</tag>
</module>

<module>
   <name>protoComponent</name>
   <parameters>--name /protoComponentB</parameters>
   <node>dream1</node>
   <tag>protoComponentB</tag>
</module>

<module>
   <name>yarpview</name>
   <parameters>--name /binaryImage</parameters>
   <node>dream1</node>
   <tag>output_image</tag>
</module>

<connection>
  <from>/robot/camera/left</from>
  <to>/protoComponentA/image:i</to>
  <protocol>tcp</protocol>
</connection>

<connection>
  <from>/protoComponentA/image:o</from>
  <to>/binaryImage</to>
  <protocol>tcp</protocol>
</connection>
```

Here we have two `protoComponent` components: `protoComponentA` and `protoComponentB` with connections from port `/robot/camera/left` to `/protoComponentA/image:i` and from `/protoComponentA/image:o` to `/binaryImage`.

The `/protoComponentB/image:i` and `/protoComponentB/image:o` ports are not connected to anything in this example.

Note again the use of the `<tag> ... </tag>` construct to provide a unique name for this particular instantiation of the YARP module (i.e. the component) . The corresponding executable to be run is identified by the `<name>...</name>` construct. All `<tag> ... </tag>` arguments must be unique.

3.5 External Configuration

YARP supports external configuration of the behaviour of the component through the use of `.ini` configuration files. This allows the behaviour of individual instances of a component to be customized by the application developer without recourse to the component developer. It is the responsibility of the component developer to expose all these parameters to the user.

YARP provides the component developer with a `ResourceFinder` class that provides methods to read and parse key-value parameters either from the `.ini` configuration file or from the arguments associated with a command-line invocation of that module. Note that if key-value arguments are supplied in the configuration file and in the command-line invocation, the latter takes precedence.

In the case of the prototype component example, `protoComponent` takes an integer parameter specified by the key-value pair `threshold <value>`. The C++ `ResourceFinder` code to parse this parameter is shown in the segment below.

```
/* get the threshold value */

thresholdValue  = rf.check("threshold",
                           Value(8),
                          "Key value (int)").asInt();
```

Similarly, the C++ `ResourceFinder` code to parse the `name` parameter described in the previous section is shown in the segment below.

```
/* get the string to form the stem of all module port names */

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

YARP assumes that every module has a default configuration file located in a default path. The `ResourceFinder` class that provides methods to set these defaults, as follows.

```
// can be overridden by --from parameter
rf.setDefaultConfigFile("protoComponent.ini");

//overridden by --context parameter
rf.setDefaultContext("protoComponent/conf");
```

3.6 Different Contexts

YARP provides a facility to allow components to run in different contexts. This means that it allows the component user to specify the location of the component's configuration files and other resources. This was hinted at in the previous example where it was suggested that both of the defaults could be overridden by two parameters, the `--from` parameter (which defines the name of the new configuration file) and the `--context` parameter (which defines a new path to search for the configuration file). The following code segment provides an example.

```
<module>
   <name>protoComponent</name>
   <parameters>--context /altPath/conf  --from altFile.ini </parameters>
   <node>dream1</node>
   <tag>protoComponent</tag>
</module>
```

More complete documentation on the facilities offered by the `ResourceFinder` class can be found here `http://wiki.icub.org/wiki/YARP_ResourceFinder`.

3.7 Runtime Configuration

The guidelines state that a CBSE-based robot programming framework should provide a facility for runtime configuration of the behaviour of the component to allow users and other components to control the component's processing, if necessary. This is provided in YARP by way of a special port with the same name as the module which can be used to change the parameter values while the module is executing.

Note that the name of this port mirrors whatever is provided by the `--name` parameter value. The port is attached to the terminal so that you can type in commands and receive replies. The port can be used by other modules but also interactively by a user through the YARP `rpc` directive, viz.: `yarp rpc /protoComponent`. This opens a connection from a terminal to the port and allows the user to then type in commands and receive replies.

The following code segment shows how the `respond()` method in the resource finder `RFModule` class handles input from this port.

```
bool protoComponent::respond(const Bottle& command, Bottle& reply)
{
   string helpMessage =  string(getName().c_str()) +
                                    " commands are: \n" +
                                    "help \n" +
                                    "quit \n" +
                                    "set thr <n> ... set the threshold \n" +
                                    "(where <n> is an integer number) \n";
   reply.clear();

   if (command.get(0).asString()=="quit") {
       reply.addString("quitting");
       return false;
   }
   else if (command.get(0).asString()=="help") {
       cout << helpMessage;
       reply.addString("command is: set thr <n>");
   }
   else if (command.get(0).asString()=="set") {
      if (command.get(1).asString()=="thr") {
           thresholdValue = command.get(2).asInt(); // set parameter value
         reply.addString("ok");
      }
   }
   return true;
}
```

Note that runtime configuration, as described above, corresponds to the Coordination concern in the Four Cs, the other three being Configuration, Computation, and Communication.

3.8 Assignment of Resources

The guidelines state that a CBSE-based robot programming framework should provide explicit assignment of resources to the component, where necessary. Again, YARP explicitly supports this through the resource finder `ResourceFinder` class by allowing the developer to read multiple configuration files, not just the configuration file named after the module itself. Furthermore, YARP provides a resource finder method to find files in a series of defined search paths, or contexts. For example, the code segment below finds the file containing an image resource that can then be read and processed.

```
imageFilename  = rf.check("imageFile",
                          Value("image.bmp"),
                          "image filename (string)").asString();

imageFilename = (rf.findFile(imageFilename.c_str())).c_str();

...

if (yarp::sig::file::read(inputImage, imageFilename->c_str())) {
   cout << "ImageSourceThread::threadInit: input image read completed"
          << endl;
   return true;
}
else {
   cout << "ImageSourceThread::threadInit: unable to read image file"
          << endl;
   return false;
}
```

3.9 Asynchronous Communication

All YARP module run and communicate asynchronously.

3.10 Distributed Computing

YARP supports distributed computing. It is possible to run components on a network of computer, be it a distributed system, a collection of PCs, or a server farm. Specifically, YARP allows an application developer (not just a component developer) to assign a component to a given compute-server.

This is achieved using an XML `<node>...</node>` construct in the examples above. This node name is the given to a uniquely-labelled YARP run server, in this case `dream1`. One YARP run-server must be running on each computer connected to the network on which YARP modules are to be run. This is accomplished simply by issuing a YARP command `PC> yarprun --server /dream1` (note the `/` prefix). Changing the name of the node changes the computer on which that module will be executed, provided a run-server is running on that computer and it has been declared to the YARP name server with the `yarprun` command. Note in passing that while there is one YARP run server running on each computer on the YARP network, there is only one YARP name-server running on some computer on the network.

```
<module>
   <name>protoComponent</name>
   <node>dream1</node>
   <tag>protoComponent</tag>
</module>

<module>
   <name>yarpview</name>
   <parameters>--name /binaryImage </parameters>
   <node>dream1</node>
   <tag>output_image</tag>
</module>

<connection>
  <from>/robot/camera/left</from>
  <to>/protoComponent/image:i</to>
  <protocol>tcp</protocol>
</connection>

<connection>
  <from>/protoComponent/image:o</from>
  <to>/binaryImage</to>
  <protocol>tcp</protocol>
</connection>
```

3.11 Stable Interfaces

Another guideline states that components should have stable interfaces. This can be ensured by enforcing backward compatability in the development of new code and the modification of legacy code.

3.12 Abstract Component Interfaces

The robot programming framework should provide facilities to support abstract component interfaces. As noted above, this effectively means that there is support for high-level communication using ports. Again, this is the very essence of YARP, for which ports are central to inter-module communication. YARP has a variety of methods to support abstraction in data representation, including the `Bottle`

class. This is a simple collection of objects that can be described and transmitted in a portable way. This class has a well-defined, documented representation in both binary and text form. Objects are stored in a list, which you can add to and access.

The `respond()` method in the resource finder `RFModule` class in the example in Section 3.7 uses bottles.

3.13 Multiple Transport Layer Protocols

YARP support a flexible communication infrastructure with multiple transport layer protocols so that the ports can be configured to use different communication protocols.

Every YARP connection has a specific type of carrier associated with it. This corresponds roughly to the "transport" used to carry data. For example there is the tcp carrier, the udp carrier, the mcast (multi-cast) carrier. For each transport, there can be several variants of carrier. For example, across tcp, there can be the basic binary-mode tcp carrier, or the text (text-mode) carrier, or the fast_tcp carrier which takes some shortcuts for speed. YARP has an extensive repertoire of carriers, including the following:

- tcp carrier
- udp carrier
- mcast (multicast) carrier
- shmem (shared memory) carrier
- local (within-process) carrier
- text (text-mode across tcp) carrier
- text_ack (text-mode across tcp with acknowledgement) carrier
- fast_tcp (tcp without acknowledgement) carrier
- http carrier

YARP also allows for the creation of entirely new kinds of carriers.

3.14 Graphic User Interface Tools

YARP provides `gyarpmanager`, a graphic user interface tool for running and managing multiple robot applications, i.e. XML files, on a set of computers. More information is available here `http://wiki.icub.org/yarpdoc/yarpmanager.html`.

4 Make File Utilities

We have decided to adopt the CMake tool to describe how the DREAM software should be compiled and linked. CMake lets you express the structure of your software in a portable way that allows it to be compiled with several of different tools. So, for example, you don't have to maintain Makefiles and Visual Studio projects and you can have a common "source" — the CMake file — that can generate either. Cmake can be downloaded from `http://www.cmake.org/HTML/Download.html`.

A CMake file will be provided in the initial release of the DREAM software respository (accessed using SVN; see below). This CMake file will initially provide what is necessary to compile two systems:

- A prototype component;
- The top-level DREAM software architecture.

The prototype component — `protoComponent` — provides examples of all the essential features of a component implemented using YARP, and, therefore, follows all the CBSE guidelines.

The top-level DREAM software architecture comprising three sub-systems, one for visual sensing and interpretation (WP4), one for child behaviour analysis (WP5), and one for robot behaviour (WP6). Each of the three components is implemented by a single component, each of which implements a port-based interface, with each of the ports either sending (producing, sourcing, publishing) or receiving (consuming, sinking, subscribing to) the abstract data corresponding to the required system functionality defined in Deliverables D3.2 and D3.3.

5 Software Libraries

On Windows, YARP requires the ACE library, a free and open source library that provides a portable operating systems interface. Download ACE here: `http://download.dre.vanderbilt.edu/`

YARP also uses the OpenCV computer vision library and the GTK library for, among other things, its YARPView image viewer. Download OpenCV here: `http://opencv.org/` and GTK here `http://www.gtk.org/`.

If any customized DREAM graphic user interfaces are required, these will use the FLTK library. FLTK can be downloaded from here `http://www.fltk.org/index.php`. An example of an FLTK interface to the prototype component `protoComponent` will also be included in the DREAM software repository.

6 Software Repository

DREAM will use Subversion (SVN) to manage version control of software and documentation on the DREAM repository.

Developers are free to choose any SVN client they prefer. TortoiseSVN is a popular Window client. It can be downloaded here `http://tortoisesvn.net/`.

The files structure of the DREAM software respository will be documented on the DREAM wiki in due course. For the moment, note that it will have directories devoted to software (source code, component-based applications, and run-time resources such as configuration and data files) and documentation (deliverables, user manuals, and reference manuals).

Part III
Standards

This part defines the project's standards governing the specification, design, documentation, and testing of all software to be developed in work-packages WP4, WP5, and WP6. Particular emphasis is placed on the latter phases of the life cycle — implementation, testing, and documentation — because these are particularly important for effective system integration and long-term support by third-party software engineers and system users.

We distinguish between *recommended standards* that reflect desirable practices and *mandatory standards* that reflect required practices. DREAM component software developers and robot application developers are stongly encouraged to adhere to the desirable practices but these standards do not form part of the criterial that will be used to decide whether or not a given component of application can be included in the DREAM software repository. That is, they do not form part of the DREAM software quality assurance process (as described in deliverable D3.3). On the other hand, the required practices *do* form part of the software quality assurance process and *a component or application will only be accepted for integration in the release version of the DREAM software if it complies with the corresponding mandatory standards.*

In creating the standards set out in this part, we have drawn from several sources. These include the GNU Coding Standards [18], Java Code Conventions [19], C++ Coding Standard [20], The EPFL BIRG Coding Standards [21], and the Doxygen User Manual [22].

1 Component and Sub-system Specification

The DREAM project aims to allow researchers and software developers as much freedom as possible in the specification of the components that meet the functional requirements for robot-enhanced therapy set out in deliverables D1.2 and D1.3. Consequently, this phase of the software development life-cycle are subject to following *recommended standards*.

Requirements
Requirements should be derived from deliverables D1.2 and D1.3 and exemplified by use cases.

Computational model
Any underlying computational model should be clearly documented.

Functional model
A functional specification should be documented, together with a functional decomposition into smaller functional units. If an object-oriented approach is being used, then the functional model should include a class and class-hierarchy definition.

Data model
The data model should be described with an entity-relationship diagram. A data dictionary should be produced, identifying the input functional, control, system configuration data, output functional, control, system configuration data for each process or thread in the component. If an object-oriented approach is being used, then an object-relationship model should be included.

Process flow model
A process flow model should be produced, e.g. a data-flow diagram (DFD), identifying data flow, control flow, and persistent data sources and sinks.

Behavioural model

A behavioural model should be produced, e.g. a state transition diagram. If an object-oriented approach is being used, then an object-behaviour model should be included.

2 Component Design

The principles of good design have already been addressed at length in Part I, in general, and in the guidelines set out in Part I, Section 5, in particular. We consider these guidelines to be a set of *recommended standards* for component design and we will not repeat them here. However, it is important to note that these guidelines give rise to several essential practices — and mandatory standards — in component implementation, as outlined in the YARP examples in Part II, Section 3. We return to these in the next section.

3 Component Implementation

Some implementation standards are mandatory, others are recommended. The mandatory standards for implementation of components are the first set of standards that form part of the software quality assurance process and the component or application will only be accepted for integration in the release version of the DREAM software if it complies with these standards.

The mandatory implementation standards include the following. Details of each set of standards are provided in an appendix to this deliverable for easy reference by developers.

1. Mandatory standards for file names and file organization (Appendix A).
2. Mandatory standards for internal source code documentation (Appendix B).
3. Mandatory standards for component functionality: The 4Cs of configuration, coordination, computation, and communication using YARP utility classes (Appendix C).

The recommended implementation standards include the following. Again, details of each set of standards are provided in an appendix to this deliverable for easy reference by developers.

1. Recommended standards for programming style (Appendix D).
2. Recommended standards on programming practice (Appendix E).

To make them easier to follow, all these standards make reference to a YARP implementation of a prototype component which will be made available on the DREAM SVN repository along with a sample application. Support documentation will be provided on the DREAM wiki.

4 Testing

DREAM software will be subject to a spectrum of test procedures, including black-box unit tests, white-box structural tests, regression tests, and acceptance tests.

4.1 Black-box Unit Tests

Black-box testing is a testing strategy that checks the behaviour of a software unit — in this case a component — without being concerned with what is going on inside the unit; hence the term "black-box". Typically, it does this by providing a representative sample of input data (both valid and invalid), by describing the expected results, and then by running the component against this test data to see whether or not the expected results are achieved.

Component software developers must provide a unit test with every component submitted for integration into the DREAM release. This test comprises two parts: an XML application and a test description.

The XML application should launch the component being tested and connect it through its ports to a data source and a data sink.

The sources and sinks can be files or driver and stub components written specifically by the developer to provide the test data.

The application file should be named after the component but with the suffix `TEST`, for example `protoComponentTEST.xml`.

Source and sink data files should be located in the `config` directory (see Appendix A) and, if they are used, driver and stub components should be located in `src` directory so that they will be compiled and linked along with the component being tested. The test application file should be located in the `app` directory.

Instructions on how to run the test should be included in a `README.txt` file, also located in the `app` directory (see Appendix A, Section A.3.2).

In general, these instructions should describe the nature of the test data and the expected results, and it should explain how these results validate the expected behaviour of the component. In particular, these instructions should explain how the four Cs of communication, configuration, computation, and coordination are exercised by the test.

Validation of communication and computation functionality will typically be established by describing the (sink) output data that will be produced from the (source) input data.

Validation of configuration functionality will typically be established by stating what changes in behaviour should occur if the values for the component parameters in the component configuration (`.ini`) file are altered (see Appendix C, Section C.1).

Validation of coordination functionality will typically be established by stating what changes in behaviour should occur when commands are issued interactively by the user to the component using the port named after the component itself (see Appendix C, Section C.2).

4.2 White-box Structural Tests

White-box testing is a testing strategy that checks the behaviour of a software unit or collection of units in a (sub-)systems by exercising all the possible executions paths within that unit or system. Thus, white-box testing different fundamentally from black-box testing in that it is explicitly concerned with what is going on inside the unit.

In DREAM we will perform white-box testing on a system-level only. Components will not be subject to white-box testing when being integrated into the DREAM software release, although developers themselves are encouraged to use white-box testing before submitting the component for integration.

White-box testing will be effected by checking 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. Thus, this white-box test will establish the validity of the inter-component communication but not its functionality (which is tested using the black-box unit test).

4.3 Regression Tests

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 compatability, 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.

4.4 Acceptance Tests

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.

5 Documentation

The primary vehicle for documentation will be the DREAM wiki. It will have sections dealing with the following five issues.

1. Software installation guide
2. Software users guide
3. Software development guide
4. Software integration guide
5. Component reference manual

The software installation guide will provide a step-by-step guide to downloading, installing, and checking the software required to develop DREAM software and write and run DREAM robot applications. It will derived from Part II — Software Development Environment — in this deliverable (D3.2), and augmented with more detailed instructions whereever they are needed.

The software users guide will explain how to use YARP and YARP tools to manage and run applications comprising a collection of components.

The software development guide will be derived directly from the relevant sections of this deliverable (D3.2), addressing the development of component software and robot applications. It will include a detailed step-by-step walkthough of the design and implementation of the example prototype component `protoComponent` used throughout this deliverable and the development of an application to use it.

The software integration guide will describe the procedures for unit testing of individual components and submitting them for integration. This guide will be derived from Part III, Section 4 of this deliverable, and augmented with more detailed instructions whereever they are needed.

The component reference manual will be derived mainly from the Doxygen documentation comments in the header file of each component; see Appendix B, Section B.2 for details of the information that is captured in this documentation.

For easy reference, the wiki will also include copies of the standards described in this deliverable.

1. Mandatory Standards for File Organization
2. Mandatory Standards for Internal Documentation
3. Mandatory Standards for Component Functionality
4. Mandatory Standards for Testing
5. Recommended Standards for Programming Style
6. Recommended Standards for Programming Practice

The mandatory standards are contained in Appendices A, B, and C (File Organization, Internal Documentation, and Component Functionality, respectively), as well as Section 4 on Testing. The recommended standards are contained in Appendices D and E (Programming Style and Programming Practice, respectively).

References

[1] M. Radestock and S. Eisenbach. Coordination in evolving systems. In *Trends in Distributed Systems, CORBA and Beyond*, pages 162—176. Springer, 1996.

[2] P. Soetens, H. Garcia, M. Klotzbuecher, and H. Bruyninckx. First established CAE tool integration. BRICS Deliverable D4.1, 2010. http://www.best-of-robotics.org.

[3] A. Shakhimardanov, J. Paulus, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar. Best practice assessment of software technologies for robotics. BRICS Deliverable D2.1, 2010. http://www.best-of-robotics.org.

[4] H. Bruyninckx. Robotics software framework harmonization by means of component composability benchmarks. BRICS Deliverable D8.1, 2010. http://www.best-of-robotics.org.

[5] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. The BRICS component model: A model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1758–1764, New York, NY, USA, 2013. ACM.

[6] C. Schlegel, A. Steck, and A. Lotz. Model-driven software development in robotics: Communication patterns as key for a robotics component model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[7] D. Brugali and P. Scandurra. Component-Based Robotic Engineering (Part I). *IEEE Robotics and Automation Magazine*, pages 84–96, December 2009.

[8] D. Brugali and A. Shakhimardanov. Component-Based Robotic Engineering (Part II). *IEEE Robotics and Automation Magazine*, pages 100–112, March 2010.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666—677, 1978.

[10] http://www.omg.org.

[11] M. Quigley, K. Conley, B.P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[12] G. Metta, P. Fitzpatrick, and L. Natale. Yarp: yet another robot platform. *International Journal on Advanced Robotics Systems*, 3(1):43–48, 2006.

[13] J. Baillie. Urbi: towards a universal robotic low-level programming language. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2005*, pages 3219—3224, 2005.

[14] P. Soetens. *A software framework for real-time and distributed robot and machine control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2006.

[15] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. *Software Engineering for Experimental Robotics*, chapter Orca: a component model and repository, pages 231—251. Springer Tracts in Advanced Robotics. Springer, 2007.

[16] A. Makarenko, A. Brooks, and T. Kaupp. On the benefits of making robotic software frameworks thin. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2007*, 2007.

[17] E. I. Barakova, J. C. C. Gillesen, B. E. B. M. Huskens, and T. Lourens. End-user programming architecture facilitates the uptake of robots in social therapies. *Robotics and Autonomous Systems*, 61:704–713, 2013.

[18] R. Stallman *et al. GNU Coding Standards*. 2005. http://www.gnu.org/prep/standards/.

[19] *Java Code Conventions*. http://java.sun.com/docs/codeconv/CodeConventions.pdf.

[20] *C++ Coding Standards*. http://www.possibility.com/Cpp/CppCodingStandard.html.

[21] *Birg Coding Standards for C/C++*. http://birg.epfl.ch/page26861.html.

[22] D. van Heesch. *Doxygen User Manual*. 2005. http://www.doxygen.org.

[23] *FLTK User Manual*. http://www.fltk.org.

Appendix A Mandatory Standards for File Organization

A.1 Directory Structure

Files for a single component should be stored in a directory named after the component, e.g. `protoComponent`. Note that we keep the leading letter in lowercase since this directory refers to a component, not a class (in which case, in most conventions, the leading letter would be in uppercase).

This directory should have three sub-directories: `src`, `app`, and `config` (see Fig. 5).

The `src` directory contains the `*.c`, `*.cpp`, and `*.h` files.

The `app` directory contains the `*.xml` XML robot application files.

The `config` directory contains the configuration file, named after the component and with a `.ini` extension. For example, `protoComponent.ini`. Other resources, e.g. image files should also be placed in the `config` directory.

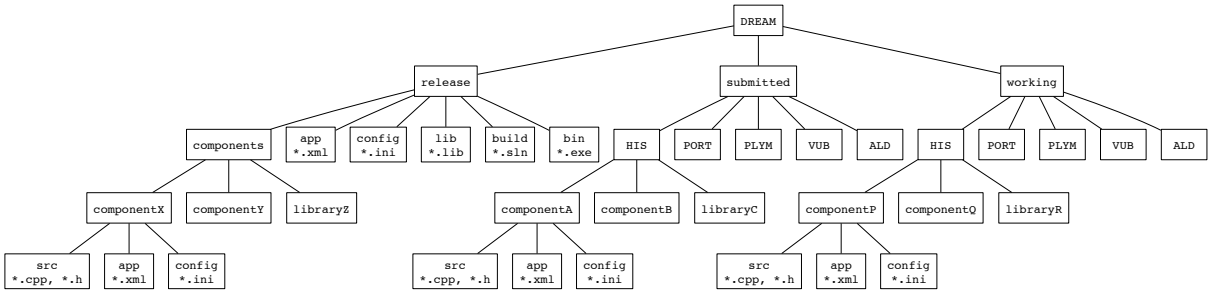

Figure 5: Partial directory structure for the DREAM software repository.

A.2 Filename Roots and Extensions

All files should have the same root, reflecting computational purpose of the component, e.g. `protoComponent`.

Source code files for C and C++ should use a `.c` and `.cpp` extension, respectively.

Header files should have a `.h` extension in both cases.

A.3 File Organization

A.3.1 Source Files

Normally, there are three different types of source code file in any given project. These are the interface, implementation, and application files.

In a CBSE project, the source code *application file* is replaced by the application script file or application XML file that runs the various components and connects them together in a working system.

The *interface file* is a header file with the class declarations and method prototype declarations but no method implementations (C++) or the function prototype declarations (C).

The *implementation file* contains the source code for the implementation of each class method (C++) or the source code of each function (C). General purpose functions might eventually be placed in a library.

In DREAM, for YARP modules we further separate the implementation into three files.

The first source code file — e.g. `protoComponentMain.cpp` — contains the `main()` function that instantiates the module object and calls the `runModule` method to effect the configuration, coordination, computation, and communication for that component.

The second source code file — e.g. `protoComponentConfiguration.cpp` — contains the code that handles the component's configuration and coordination functionality.

The third source code file — e.g. `protoComponentComputation.cpp` — contains the code that handles the component's computation and communication functionality.

In summary, the implementation of a typical component will comprise four files.
Using the `protoComponent` component as an example, these are:

1. `protoComponent.h`
2. `protoComponentMain.cpp`
3. `protoComponentConfiguration.cpp`
4. `protoComponentComputation.cpp`

All four should be placed in the `src` directory.

A.3.2 Application Files

The `app` directory should contain at least one XML application file. It should be named after the component but with the suffix `TEST`, for example `protoComponentTEST.xml`. This application file will be used to validate that the component works correctly and will be used to test the component when it is being submitted for integration. Instructions on how to run the test should be included in a `README.txt` file in the same directory.

A.3.3 Configuration Files

Each component must have an associated configuration file, named after the component,
e.g `protoComponent.ini` . It is stored in the `config` directory.

The configuration file contains the key-value pairs that set the component parameters. For readability, each key-value pair should be written on a separate line.

Appendix B Mandatory Standards for Internal Source Code Documentation

B.1 General Guidelines

Two types of comments are required: documentation comments and implementation comments. Implementation comments are those which explain or clarify some aspect of the code. Documentation comments are intended to be extracted automatically by the Doxygen tool to create either HTML or LaTeX documentation for the program. Documentation comments describe the functionality of a component from an implementation-free perspective. They are intended to be read by developers who won't necessarily have the source code at hand. Thus, documentation comments help a developer understand how to use the component through its application programming interface (API), rather than understand its implementation.

Documentation comments are delimited by `/** ... */`.

Implementation comments are delimited by `/* ... */` and `//`.

Implementation comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.

Information about how the executable should be compiled and linked or in what directory it resides should not be included as a comment. This information should go in the `README.txt` file.

All comments should be written in English.

B.2 Documentation Comments

Documentation comments often describe classes, constructors, destructors, methods, members, and functions. However, in DREAM we will use it to provide external documentation on the 4Cs: component functionality (i.e. computation), configuration, communication (through ports), and coordination (through ports, including the module-name port).

The Doxygen documentation system [22] is used to extract the documentation comments and create external documentation in HTML or LaTeX. Although Doxygen supports several documentation formats, we will stick to the Javadoc format as it is widely-accepted and it facilitates visually-pleasing and unobstrusive comments.

Each documentation comment is set inside the comment delimiters `/** ... */`. Within this comment, several keywords are used to flag specific types of information. We will treat each of these below by way of an example taken from the prototype component module, specifically the documentation comments in `protoComponent.h`. Since we are not using Doxygen documentation comments to described classes, there are no documentation comments in the in the three `*.cpp` files.

Note that to use the JavaDoc style `JAVADOC_AUTOBRIEF` must be set to `YES` in the Doxygen configuration file.

Note also that blank lines are treated as paragraph separators and the resulting documentation will automatically have a new paragraph whenever a blank line is encountered in a documentation comment.

The First Documentation Comment

All source files that contain general documentation comments (rather than class documentation comments) must begin with a documentation comment that identifies the file being documented, as follows.

```
/** @file <filename> <one line to identify the nature of the file>
 *
 * Version information
 *
 * Date
 */
```

This applies in particular to the `<componentName>.h` file where the component API is documented.

The following is a list of the mandatory sections for which documentation comments must be provided. These are taken from the example `protoComponent.h` file and you should refer to the full listing of this file in the DREAM SVN repository (see Part II, Section 6 and Appendix A, Section A.3) and to the software development guidelines on the DREAM wiki (see Part III, Section 5).

```
/**
 * @file protoComponent.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>
 *
```

```
 *  - \c /protoComponent \n

 ...

 * <b>Output ports</b>
 *
 *  - \c /protoComponent \n

...

 * <b>Port types </b>

 ...

 * \section in_files_sec Input Data Files

 ...

 * \section out_data_sec Output Data Files

 ...

 * \section conf_file_sec Configuration Files
 *
 * \c protoComponent.ini

 ...

 * \section tested_os_sec Tested OS

 ...

 * \section example_sec Example Instantiation of the Module
 *
 * <tt>protoComponent --name <componentName>
 *                    --context components/<componentName>/config
 *                    --from <componentName>.ini </tt>

 ...

 * \author
 * <forename> <surname>
 *
 * Copyright (C) 2014 DREAM Consortium
 *
 */
```

B.3 Implementation Comments

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures, and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

The First Block Comment

All source files should 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.
 */
```

This text is not included in a documentation comment (see below) because we don't want it to be extracted into the documentation by Doxygen. The comment should be placed at or close to the beginning of every source file: `*.h`, `*.cpp`, and `*.c`.

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format.

```
if (condition) {

   /* Handle the condition. */

   ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a segment of code, they should all be indented to the same level.

```
if (a == b) {
   return TRUE;                 /* special case */
}
else {
   return general_answer(a);    /* only works if a != b */
}
```

End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments. However, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow.

```
if (foo > 1) {

   // look left
   ...
}
else {
   return false;  // need to explain why
}

//if (foo > 1) {
//
//   // look left
//   ...
//}
//else {
//   return false;  // need to explain why
//}
```

Appendix C Mandatory Standards for Component Functionality

The standards set out in this section are concerned with adherence to the principles of the 4Cs of Configuration, Coordination, Computation, and Communication discussed in Part I. By their very nature, these standards are closely tied to the implementation of the software and the YARP support for that implementation. However, this is not an appropriate place to provide a detailed guide to component-based software development and that will be documented in due course on the DREAM wiki. On the other hand, we will make detailed reference to specific YARP-based implementation techniques in the following in order to make clear that the standards entail both abstract requirements and specific implementation techniques. To see how all this works together, you should refer to the complete example implementation of a prototypical component `protoComponent` on the DREAM wiki.

For the moment, please take careful note of two important issues when it comes to the implementation of a component with YARP.

First, to develop a component for DREAM you need to define two derived classes:

- The first class is derived from the `yarp::os::RFModule` class,
  e.g. `class ProtoComponent : public RFModule {}`
- The second class is derived from either `yarp::os::Thread` or `yarp::os::RateThread`,
  e.g. `class ProtoComponentThread : public Thread {}`

The first derived class takes care of the first two Cs in the CPC model (Configuration and Coordination) and is defined in the `protoComponentConfiguration.cpp` file.

The second derived class takes care of the second two Cs in the CPC model (Computation and Communication) and is defined in the `protoComponentComputation.cpp` file.

Both derived classes are declared in `protoComponent.h`.

The `ProtoComponent` class is instantiated as a object in `protoComponentMain.cpp`:

```
/* create your module */

ProtoComponent protoComponent;
```

The derived class methods for `ProtoComponent` are defined in `protoComponentConfiguration.cpp`.

The `ProtoComponentThread` class (or the `ProtoComponentRateThread` class) is instantiated as a object and processing started in the `ProtoComponent::configure()` method in the configuration `.cpp` file, e.g. `protoComponentConfiguration.cpp`, as follows.

```
/* create the thread and pass pointers to the module parameters */

protoComponentThread = new ProtoComponentThread(&imageIn,
                                                &imageOut, &thresholdValue);

/* now start the thread to do the work */

protoComponentThread->start();
```

The derived class methods for `ProtoComponentThread` or `ProtoComponentRateThread` are defined in `protoComponentComputation.cpp`.

Second, you need to instantiate a `ResourceFinder` class, e.g. `ResourceFinder rf`. The `rf` object is instantiated in `protoComponentMain.cpp` and the object methods are used to prepare and configure the resource finder, identifying

1. where the resource finder should look for the information that defines the root of the path to use when searching for the component configuration files and resources,
2. the default context (i.e. the path default to be appended to the root path), and
3. the default name of the configuration file.

This is explained further in Sections C.1.1 – C.1.4, as well as on the DREAM wiki.

C.1 Component Configuration Standards

A component is configured by its parameters. These are defined either

1. in a configuration file, or
2. in the application file in the `<parameter></parameter>` section, or
3. by the command line arguments.

Every component must define a default configuration file and default path to search for that file (known as a *context*). Furthermore, every component must allow the use to override these defaults with the component parameters.

In the following, we cover the standards for handling these default and user-defined values first, before proceeding to cover standards for handling parameters in general.

C.1.1 Default Configuration File

YARP assumes that every component has a default configuration file. For the purposes of these standards, this configuration file must have a `.ini` extension and it must be named after the component, e.g. `protoComponent.ini`.

A component must set this default filename.

This is accomplished with the `setDefaultConfigFile()` method in the `ResourceFinder` class, as follows.

```
/* can be overridden by --from parameter */

rf.setDefaultConfigFile("protoComponent.ini");
```

This code goes in the main `.cpp` file, e.g. `protoComponentMain.cpp`.

C.1.2 Default Configuration Context

YARP assumes that the default configuration file is located in a default path. A component must set this default path.

This is accomplished with the `setDefaultContext()` method in the `ResourceFinder` class, as follows.

```
/* overridden by --context parameter */

rf.setDefaultContext("protoComponent/config");
```

This code also goes in the main `.cpp` file, e.g. `protoComponentMain.cpp`.

Note, however, that this context is *not* the full path where YARP should search for the configuration file. The full path is formed in flexible but rather complicated manner which we will now explain.

YARP uses a policy file to determine the paths to search for configuration files. This requires you to tell YARP where to look for the information required to initialize the root of the path to use when searching for component configuration files (in fact, YARP allows for several default paths, as described below). This is accomplished with the `configure()` method in the `ResourceFinder` class, as follows.

```
rf.configure("DREAM_ROOT", argc, argv);
```

`DREAM_ROOT` is an environment variable (which you need to define and initialize). Its value is the path where a configuration file named `DREAM_ROOT.ini` is located (e.g. `C:\DREAM`). This `.ini` file contains the YARP policies for finding DREAM configuration files. It defines two parameters `capability_directory` and `default_capability`.

`capability_directory` has one value associated with it: the path that is appended to the value of the `DREAM_ROOT` environment variable. For example, if we had

```
capability_directory Release
```

then the default path would *begin*

```
C:/DREAM/Release
```

(by concatenating values associated with `DREAM_ROOT` and `capability_directory`.

However, YARP will search several directories based on this partial path. It forms the full path to search for the configuration files by appending the *context*, i.e. the value(s) associated with the `default_capability` parameter key. For example, if we had

```
default_capability configuration components/config
```

then the paths to be searched would be

```
C:/DREAM/Release/config
C:/DREAM/Release/components/config
```

YARP also searches one other path (and, in fact, it searches it first). This path is the one that is formed by appending the value of the default context provided as an argument of the `rf.setDefaultContext()` method (described above) or the value of the `--context` parameter in an application. For example, if a component set the default context as follows

```
rf.setDefaultContext("components/protoComponent/config");
```

or the application specified

```
--context components/protoComponent/config
```

then YARP would first search for the configuration file in

```
C:/DREAM/Release/components/protoComponent/config
```

before then searching

```
C:/DREAM/Release/config
C:/DREAM/Release/components/config
```

as dictated by the YARP policies in the `DREAM_ROOT.ini` configuration file.

C.1.3 User-defined Configuration File

A component must allow the default name of the configuration file to be overridden by the configuration time. This is accomplished with a the `--from` parameter. You don't have to do anything to implement this functionality as the `ResourceFinder` class does it automatically.

C.1.4 User-defined Context

A component must allow the default context of the configuration file to be overridden by the configuration time. This is accomplished with a the `--context` parameter. You don't have to do anything to implement this functionality as the `ResourceFinder` class does it automatically.

C.1.5 Component Parameters

As stated above, a component must read its key-value parameters from a `.ini` configuration file named after the component, e.g. `protoComponent.ini`.

A component must also read its key-value parameters from the list of command line arguments.

It should do this using the `check()` method in the `ResourceFinder` class to do this. For example, the C++ `ResourceFinder` code to parse an example parameter is shown in the segment below.

```
thresholdValue  = rf.check("threshold",                 // parameter key
                           Value(8),                    // default value
                          "Key value (int)").asInt(); // key value type
```

This code is to be inserted in the `configure()` method in the configuration `.cpp` file, e.g. `protoComponentConfiguration.cpp`.

C.1.6 Component Port Names

A component must allow the port names to be set and overridden. This is treated in the same way to the parameter value above using the port name key-value parameters in the `.ini` configuration file. For example, in the following code segment, the parameter keys `protoInputPort` and `protoOutputPort` are used to read the user-defined port names, defaulting to `/image:i` and `/image:o` if they are not specified.

```
/* get the name of the input and output ports, automatically prefixing */
/* the module name by using getName()                                  */

inputPortName          = "/";
inputPortName         += getName(
                         rf.check("protoInputPort",
                         Value("/image:i"),
                         "Input image port (string)").asString()
                         );

outputPortName         = "/";
outputPortName        += getName(
                         rf.check("protoOutputPort",
                         Value("/image:o"),
                         "Output image port (string)").asString()
                         );
```

Note the convention to have `:i` and `:o` suffixes on the input and output port names, respectively.

Note also the leading `/` on all port names.

C.1.7 Component Name

A component must allow the default name of the component to be set and overridden. This is accomplished with the `--name` parameter. It should do this using the `check()` method in the `ResourceFinder` class to do this, as shown below.

```
/* get the module name which will form the stem of all module port names */

moduleName             = rf.check("name",
                         Value("protoComponent"),
                         "module name (string)").asString();
/*
 * before continuing, set the module name before getting any other parameters,
 * specifically the port names which are dependent on the module name
 */

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

C.2 Component Coordination Standards

A component must provide a for runtime configuration, i.e. coordination, of the behaviour of the module by allowing commands to be issued on a special port with the same name as the component. These commands will typically alter the parameter values while the module is executing.

As already noted, the name of this port mirrors whatever is provided by the `--name` parameter value. The port is attached to the terminal so that you can type in commands and receive replies. The port can be used by other modules but also interactively by a user through the YARP `rpc` directive, viz.: `yarp rpc /protoComponent`. This opens a connection from a terminal to the port and allows the user to then type in commands and receive replies.

The following code segment shows how the `respond()` method in the resource finder `RFModule` class handles input from this port.

This code is to be included in the configuration `.cpp` file,
e.g. `protoComponentConfiguration.cpp`.

```
bool protoComponent::respond(const Bottle& command, Bottle& reply)
{
   string helpMessage =  string(getName().c_str()) +
                                 " commands are: \n" +
                                 "help \n" +
                                 "quit \n" +
                                 "set thr <n> ... set the threshold \n" +
                                 "(where <n> is an integer number) \n";
   reply.clear();

   if (command.get(0).asString()=="quit") {
       reply.addString("quitting");
       return false;
   }
   else if (command.get(0).asString()=="help") {
       cout << helpMessage;
       reply.addString("command is: set thr <n>");
   }
   else if (command.get(0).asString()=="set") {
      if (command.get(1).asString()=="thr") {
           thresholdValue = command.get(2).asInt(); // set parameter value
         reply.addString("ok");
      }
   }
   return true;
}
```

C.3 Component Computation Standards

A component must implement the functional aspect of the code using either a YARP `yarp::os::Thread` class or `yarp::os::RateThread` class. As already stated at the beginning of this section:

- The derived class is declared in the `.h` file.
- It is instantiated as a object and processing started in the overloaded `RFModule` method `configure()` in the configuration file, e.g. `protoComponentConfiguration.cpp`, viz.

```
protoComponentThread = new ProtoComponentThread(&imageIn,
                                                &imageOut,
                                                &thresholdValue);
protoComponentThread->start(); // this calls threadInit() and
                               // it if returns true, it then calls run()
```

- The derived class methods are defined in `protoComponentComputation.cpp`.

All parameters read and initialized in the `configure()` method of the `yarp::os::RFModule` class must be passed explicitly when instantiating the derived `yarp::os::Thread` or `yarp::os::RateThread` object. For example, `imageIn`, `imageOut`, and `thresholdValue` arguments above are all initialized with the `rf.check()` method.

C.4 Component Communication Standards

A component must effect all communication with other component using YARP ports. Examples of port usage are provided in the `protoComponent` example on the DREAM wiki.

Appendix D Recommended Standards for Programming Style

D.1 Indentation and Line Breaks

Either three or four spaces should be used as the unit of indentation. Choose one standard and stick to it throughout the code.

Do not use tabs to indent text. If you are using Microsoft Visual C++ turn off the tabs option (go to `Tools -> Options -> Text Editor -> C/C+ -¿ Tabs+` and click the `insert spaces` radio button; you can set the number of spaces to three or four here too).

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools.

When an expression will not fit on a single line, break it according to the following general principles.

- Break after a comma.
- Break before an operator.
- Align the new line with the beginning of the expression at the same level on the previous line.

For example, consider the following statements.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longName6;  // Good break

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longName6;  // bad break: avoid
```

D.2 Declarations

Number Per Line
One declaration per line is recommended since it encourages commenting:

```
int level; // indentation level
int size;  // size of table
```

is preferable to:

```
int level, size;
```

Do not put different types on the same line:

```
int foo, fooarray[]; //WRONG!
```

Initialization
Initialize local variables where they are declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

D.3 Placement

Put declarations only at the beginning of blocks. A block is any code surrounded by curly braces { and }. Don't wait to declare variables until their first use. Ideally, declare all variables at the beginning of the method or function block.

```
void myMethod() {
   int int1 = 0; // beginning of method block

   if (condition) {
      int int2 = 0; // beginning of "if" block
      ...
   }
}
```

Class Declarations

The following formatting rules should be followed:

- No space between a method name and the parenthesis (starting its parameter list.
- The open brace { appears at the end of the same line as the declaration statement.
- The closing brace } starts a line by itself indented to match its corresponding opening statement.

```
class Sample {
   ...
}
```

- Methods are separated by a blank line.

D.4 Statements

Simple Statements

Each line should contain at most one statement. For example:

```
argv++;          // Correct
argc++;          // Correct
argv++; argc--; // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces { statements }. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

```
if (condition) {
   a = b;
}
else {
   a = c;
}
```

`return` **Statements**
A return statement with a value should not use parentheses unless they make the return value more obvious in some way. For example:

```
return;

return myDisk.size();

return TRUE;
```

`if`, `if-else`, `if else-if else` **Statements**
The `if-else` class of statements should have the following form:

```
if (condition) {
   statements;
}

if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else {
   statements;
}
```

Always use braces `{ }`, with `if` statements. Don't use

```
if (condition) //AVOID!
   statement;
```

`for` **Statements**
A `for` statement should have the following form:

```
for (initialization; condition; update) {
   statements;
}
```

`while` **Statements**
A `while` statement should have the following form:

```
while (condition) {
   statements;
}
```

`do-while` **Statements**
A `do-while` statement should have the following form:

```
do {
   statements;
} while (condition);
```

`switch` **Statements**
A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
   statements;
   /* falls through */
case DEF:
   statements;
   break;
case XYZ:
   statements;
   break;
default:
   statements;
   break;
}
```

Every time a case falls through (*i.e.* when it doesn't include a `break` statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

D.5 Naming Conventions

C vs. C++
Naming conventions make programs more understandable by making them easier to read. Since DREAM software uses both the C language and the C++ language, sometimes using the imperative programming and object-oriented programming paradigms separately, sometimes using them together, we will adopt two different naming conventions, one for C and the other for C++. The naming conventions for C++ are derived from the JavaDoc standards [19].

C++ Language Conventions

The following are the naming conventions for identifiers when using C++ and the object-oriented paradigm.

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized	`class ImageDisplay` `class MotorController`
Methods	Method names should be verbs, in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int grabImage()` `int setVelocity()`

Variables	variable names should be in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int i;`
		`float f;`
		`double pixelValue;`

Constants	The names of variables declared as constants should be all uppercase with words separated by underscores _	`const int WIDTH = 4;`
Type Names	Typedef names should use the same naming policy as that used for class names	`typedef uint16 ComponentType`

Enum Names	Enum names should use the same naming policy as that used for class names.
	Enum labels should should be all uppercase with words separated by underscores _
		`enum PinState {`
		`  PIN_OFF,`
		`  PIN_ON`
		`};`

C Language Conventions

The following are the naming conventions for identifies when using C and the imperative programming paradigm.

Identifier Type	Rules for Naming	Examples
Functions	Function names should be all lowercase with words separated by underscores _	`int display_image()` `void set_motor_control()`

Variables	variable names should be all lowercase with words separated by underscores _ of each internal word capitalized	`int i;`
		`float f;`
		`double pixel_value;`

Constants	Constants should be all uppercase with words separated by underscores _	`#define WIDTH 4`
`#define` and Macros	`#define` and macro names should all uppercase with words separated by underscores _	`#define SUB(a,b) ((a) - (b))`

D.6 And Finally: Where To Put The Opening Brace {

There are two main conventions on where to put the opening brace of a block. In this document, we have adopted the JavaDoc convention and put the brace on the same line as the statement preceding the block. For example:

```
class Sample {
   ...
}


while (condition) {
   statements;
}
```

The second convention is to place the brace on the line below the statement preceding the block and it indent it to the same level. For example:

```
class Sample
{
   ...
}


while (condition)
{
   statements;
}
```

If you really hate the JavaDoc format, use the second format, but be consistent and stick to it throughout your code.

Appendix E Recommended Standards for Programming Practice

E.1 C++ Language Conventions

Access to Data Members
Don't make any class data member public without good reason.

One example of appropriate public data member is the case where the class is essentially a data structure, with no behaviour. In other words, if you would have used a struct instead of a class, then it's appropriate to make the class's data members public.

E.2 C Language Conventions

Use the Standard C syntax for function definitions:

```
void example_function (int an_integer, long a_long, short a_short)
...
```

If the arguments don't fit on one line, split the line according to the rules in Section **??**:

```
void example_function (int an_integer, long a_long, short a_short,
                       float a_float, double a_double)
...
```

Declarations of external functions and functions to appear later in the source file should all go in one place near the beginning of the file (somewhere before the first function definition in the file), or else it should go in a header file.

Do not put `extern` declarations inside functions.

E.3 General Issues

Use of Guards for Header Files
Include files should protect against multiple inclusion through the use of macros that guard the file. Specifically, every include file should begin with the following:

```
#ifndef FILENAME_H
#define FILENAME_H

...  header file contents go here

#endif /* FILENAME_H */
```

In the above, you should replace `FILENAME` with the root of the name of the include file being guarded *e.g.* if the include file is `cognition.h` you would write the following:

```
#ifndef COGNITION_H
#define COGNITION_H

...  header file contents go here

#endif /* COGNITION_H */
```

Conditional Compilation
Avoid the use of conditional compilation. If your code deals with different configuration options, use a conventional `if-else` construct. If the code associated with either clause is long, put it in a separate function. For example, please write:

```
if (HAS_FOO) {
    ...
}
else {
    ...
}
```

instead of:

```
#ifdef HAS_FOO
    ...
#else
    ...
#endif
```

Writing Robust Programs
Avoid arbitrary limits on the size or length of any data structure, including arrays, by allocating all data structures dynamically. Use `malloc` or `new` to create data-structures of the appropriate size. Remember to avoid memory leakage by always using `free` and `delete` to deallocate dynamically-created data-structures.

Check every call to `malloc` or `new` to see if it returned `NULL`.

You must expect `free` to alter the contents of the block that was freed. Never access a data structure after it has been freed.

If `malloc` fails in a non-interactive program, make that a fatal error. In an interactive program, it is better to abort the current command and return to the command reader loop.

When static storage is to be written during program execution, use explicit C or C++ code to initialize it. Reserve C initialize declarations for data that will not be changed. Consider the following two examples.

```
static int two = 2; // two will never alter its value
...
static int flag;
flag = TRUE;        // might also be FALSE
```

Constants
Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Variable Assignments
Avoid assigning several variables to the same value in a single statement. It is hard to read.

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

```
if (c++ = d++) { // AVOID!
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler.

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

Parentheses
Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others — you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)      // AVOID!

if ((a == b) && (c == d)) // USE
```

Standards for Graphical Interfaces
When you write a program that provides a graphical user interface (GUI), you should use a cross-platform library. The FLTK GUI library [23] satisfies this requirement.

Error Messages
Error messages should look like this:

```
function_name: error message
```

Copyright Messages

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
<Program name and version>

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.
```