CSSR for

Culturally Sensitive Social Robotics for Africa

D3.2 Software Engineering Standards Manual

Due date: **30/09/2023**
Submission Date: **26/10/2023**
Revision Date: **14/04/2025**

Start date of project: **01/07/2023** Duration: **36 months**

Lead organisation for this deliverable: **Carnegie Mellon University Africa**

Responsible Person: **D. Vernon** Revision: **1.14**

Project funded by the African Engineering and Technology Network (Afretec) Inclusive Digital Transformation Research Grant Programme		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including Afretec Administration)	
RE	Restricted to a group specified by the consortium (including Afretec Administration)	
CO	Confidential, only for members of the consortium (including Afretec Administration)	

Executive Summary

Deliverable D3.2 is a reference manual of software engineering standards. It is based on the software engineering standards manual that was written for the EU-funded DREAM project [1], Deliverable D3.2 [2], which was in turn based on the software engineering standards manual that was written for the EU-funded RobotCub project [3]. The responsible person identified on the cover of the current manual was also responsible for these two prior manuals.

The manual 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 material in Part II is partly the result of work done in Task 3.1 (System Architecture Design) which deals with the design of the CSSR4Africa system architecture. On the other hand, the standards set out in Part III are the outcome of preliminary work in Task 3.4 (Software Integration and Quality Assurance Manual), which focusses on the procedures by which software developed in Africa 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.

Contents

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 and WP5. Specification standards address functional definition, data representation, and component & subsystem behaviour. Design standards focus on the decoupling of functional computation, component communication, external component configuration, and inter-component coordination [4]. Software test strategies will include black-box unit testing, system testing on the basis of the required behaviours specified by the interaction scenarios defined in work package WP2, and regression testing for to ensure backward compatibility.

To facilitate efficient implementation and reuse of robot software by application developers, the CSSR4Africa project has decided to adopt the principles of a best-practice component-based software engineering model, such as the BRICS Component Model (BCM) [5, 6, 7, 8]. 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 and WP5, 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” [10, 11], complemented by material on the BRICS Component Model by Herman Bruyninckx and colleagues [8], and the paper by Christian Schlegel and colleagues on model-driven software development in robotics [9], 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

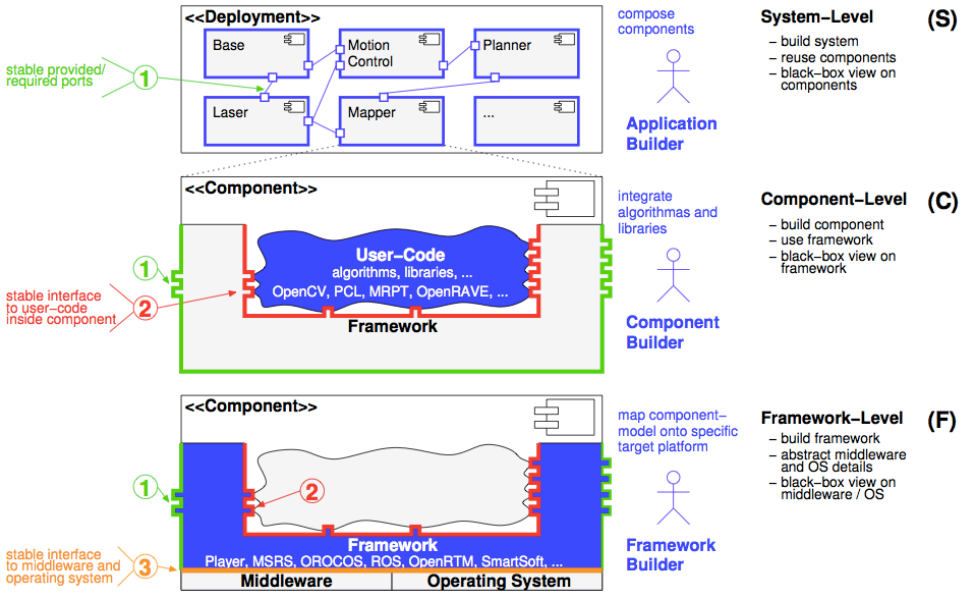

Figure 1: The relationship between the robot software system (implemented as a network of components), the robot programming framework (or platform), and the underlying middleware and operating system(s) (from [9]). The existence of stable interfaces is a key attribute of each level.

to the classic concept of communicating sequential processes (CSP) [12], 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" [7].

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 [7].

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

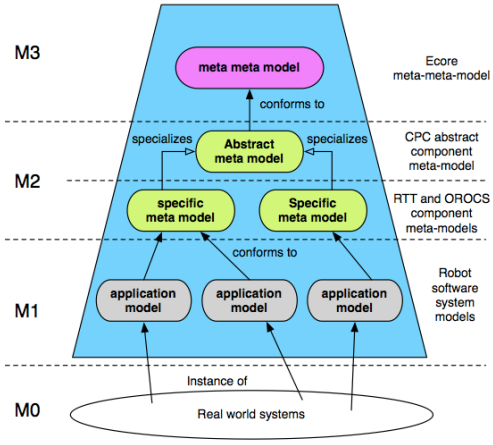

Figure 2: The four levels of abstraction from the OMG [13] applied in the BRICS Component Model (from [8]).

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 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) [13] defines four levels of model

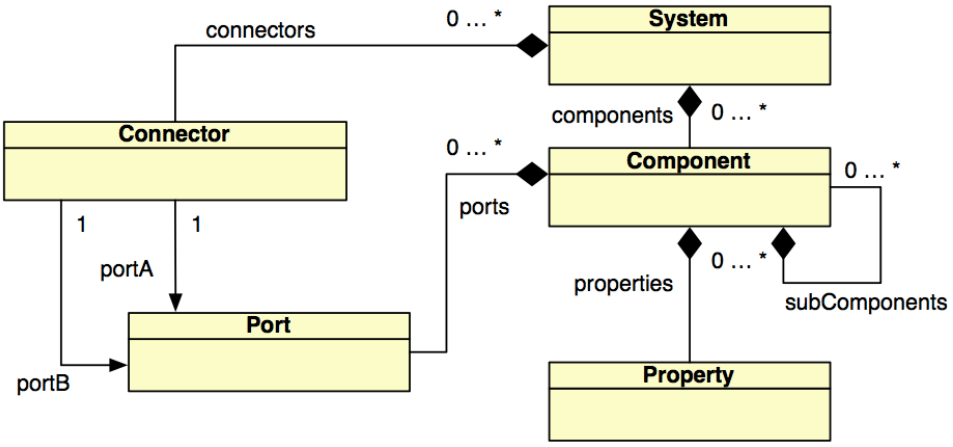

Figure 3: A UML diagram representing the Component-Port-Connector meta-model (from [8]).

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 [8] (see also Fig. 2).

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.
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 3 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 always 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.

4 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. Components 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 CSSR4Africa 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 choice of robot programming framework is based directly on the fourteen guidelines set out in the previous section. Following all guidelines is the ideal. The need to make choices based on existing competences shared by all partners in the consortium may preclude following some guidelines.

A complete set of instructions on how to download and install all the various utilities, libraries, and tools comprising the CSSR4Africa software development environment will be provided in Deliverable D3.2 Software Installation Manual.

1 Programming languages and compilers

CSSR4Africa software should be written in either the C language, the C++ language, or, where necessary, Python.

2 Operating Systems

Software will be developed for Ubuntu 20.04.

3 Robot Programming Framework

There are many robot programming frameworks. These include ROS [14], YARP [15], URBI [16], and Orca [17, 18]. All have their respective advantages and disadvantages, as explained in various surveys and comparisons [19, 20].

Our goal is to select one that will allow the software required to deliver the CSSR4Africa s functionality to be developed efficiently and effectively, bearing in mind constraints of shared partner competences. As we have already stated, we have adopted the principles of component-based software engineering (CBSE) and fourteen 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 ROS [21] as the framework for CSSR4Africa. ROS is a popular choice in many projects, despite having been criticized in the context of Component Based Software Engineering for lacking an abstract component model [9]. It has the distinct advantage that both partners in the CSSR4Africa consortium have adopted ROS for their research and teaching in robotics.

In the following, we use the terms *component* and *node* interchangeably. In general, we use *component* when referring to the abstract CBSE view of the entity and *node* when referring to the ROS implementation of a component. Similarly, we use the terms *port* and *publisher/subscriber/service/action* interchangeably, the former when referring to the abstract CBSE view of the entity and latter when referring to the ROS implementation. Finally, we use the terms *connector* (CBSE) and *topic* (ROS) interchangeably. Messages are exchanged by publishers and subscribers using ROS topics. Since communication on ROS topics is non-blocking and open-loop, i.e., a broadcast form of communication between ROS nodes, ROS provides a blocking mechanism, called a *service* that provides a two-way communication between nodes. ROS also provides a client-server mechanism, called an *action*, that allows status information to be provided by a server node to a client node while the requested operation is being carried out.

We now address the extent to which ROS adheres to the fourteen CBSE guidelines in Part I, Section 4.

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

ROS 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 ROS node corresponds to a component and it can have an arbitrary number of publishers and subscribers (ports) communicating using topics (connectors).

3.2 Robot Applications

ROS supports the implementation of robot applications by identifying the components (nodes) to be instantiated and specifying the connections (topics) between their port interfaces, i.e, using publishers and subscribers (or, possibly, services and actions).

3.3 Coarse-grained Functionality

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

3.4 Multiple Instances of Components

ROS supports the instantiation of **multiple instances** of the same component (node). This is achieved using a `__name` key when running or launching the node. ROS also allows topic names to be remapped.

3.5 External Configuration

It is accepted best practice in software engineering to ensure that the behaviour of a component can be configured externally, without having to change and recompile the code. ROS does not have specific tools to effect this, so we will simply adopt the practice of supporting external configuration of the behaviour of the component through the use of `.ini` configuration files, comprising a set of one or more key-value parameters, e.g., `threshold <value>`. 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.

3.6 Different Contexts

ROS provides a facility to allow components to run in different workspaces.

3.7 Runtime Configuration

ROS does not provide tools to support runtime configuration. If this is required, it can be effected by periodically invoking the function or method that reads the node configuration file, re-initializing the parameter values accordingly at runtime.

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, ROS does not provide tools to support this but such functionality can be accomplished through the configuration file.

3.9 Asynchronous Communication

All ROS nodes run and communicate asynchronously.

3.10 Distributed Computing

ROS supports distributed computing. It is possible to run components on a network of computers, 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.

3.11 Stable Interfaces

This 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 ROS, for which publishers, subscribers, services, and actions are central to inter-node communication.

3.13 Multiple Transport Layer Protocols

ROS supports a flexible communication infrastructure with multiple transport layer protocols so that the topics can be configured to use different communication protocols, specifically `TCPROS` and `UDPROS`.

3.14 Graphic User Interface Tools

ROS uses the `rqt` metapackage. This is a Qt-based framework for GUI development for ROS.

4 Make File Utilities

ROS uses `catkin` to manage and build packages, i.e., collections of nodes. `catkin` uses `CMake` to provide the build functionality.

5 Software Repository

The release version of all CSSR4Africa software will be uploaded to the CSSR4Africa repository on GitHub `https://github.com/cssr4africa/cssr4africa`. Software will only be uploaded after it has passed the integration checks set out in Deliverable D3.4 System Integration and Quality Assurance Manual. These checks enforce adherence to the mandatory standards in Appendix A and Appendix B of this deliverable.

Part III
Standards

This part of the document defines the project's standards governing the specification, design, documentation, and testing of all software to be developed in work-packages WP4 and WP5. 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. CSSR4Africa 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 checks that will be used to decide whether or not a given component of application can be included in the CSSR4Africa software repository, as set out in the CSSR4Africa software quality assurance process described in Deliverable D3.4. 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 CSSR4Africa 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 [22], Java Code Conventions [23], C++ Coding Standard [24], EPFL BIRG Coding Standards [25], and Python Enhancement Proposals standards [26].

1 Component and Sub-system Specification

The CSSR4Africa 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 the culturally sensitive social robotics scenario set out in deliverables D2.2 and D2.3. Consequently, this phase of the software development life-cycle are subject to following *recommended standards*.

Requirements
Requirements should be derived from deliverables D2.2 and D2.3 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 4, 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. 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 form part of the software quality assurance process and the component will only be accepted for integration in the release version of the CSSR4Africa software if it complies with these standards.

The *mandatory* implementation standards include the following.

1. Mandatory standards for file names and file organization (Appendix A).
2. Mandatory standards for internal source code documentation (Appendix B).

The *recommended* implementation standards include the following.

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

4 Testing

CSSR4Africa software will be subject to a spectrum of test procedures, including black-box unit tests, system tests, and regression 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 up to three unit tests with every component submitted for integration into the CSSR4Africa release. These tests comprises a launch file and a test description. Both should be located in the `launch` directory (see Appendix A). The three tests are either the

physical robot, the simulator, or a driver/stub test harness.

The launch files should launch the node being tested and, depending on which test is being run, the physical robot, the simulator, or a driver node to source (publish) test data, and a stub node to sink (subscribe to) the output of the node being tested.

The launch files should be named after the component but with the filename extension `.launch`. For example: `exampleComponentLaunchRobot.launch`, `exampleComponentLaunchSimulator.launch`, and `exampleComponentLaunchTestHarness.launch`.

Instructions on how to run the tests should be included in a `README.md` file, also located in the `launch` directory (see Appendix A). 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.

4.2 System Tests

A set of system tests will launch a subset of the components (nodes) in the system architecture to check that they provide the required functionality. The final two system tests will lauch the entire system and run the two user interaction scenarios.

4.3 Regression Tests

Regression testing refers to the practice of re-running all tests periodically to ensure that no unintentional changes has been introduced during the ongoing development of the CSSR4Africa software release. These tests check for backward compatability, ensuring that what used to work in the past still works now. Regression tests will be carried out on all software in the CSSR4Africa release every three months.

5 Documentation

The primary vehicle for documentation will be the CSSR4Africa wiki. It will have sections dealing with the following five issues.

1. Software installation guide.
2. Software development guide (based on this deliverable, i.e., Deliverable D3.3).
3. Software integration guide (based on Deliverable D3.4).
4. Component reference manual.

The software installation guide will provide a step-by-step guide to downloading, installing, and checking the software required to develop CSSR4Africa software and run CSSR4Africa robot applications. It will derived from Deliverable D3.3.

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.

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, from Deliverable D3.4, and augmented with more detailed instructions whereever they are needed.

The component reference manual will be derived from the contents of the node `README.md` file (see Appendix A, Section A.3) and the first documentation comment in the node application file (see Appendix B, Section B.2).

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. Recommended Standards for Programming Style
4. Recommended Standards for Programming Practice

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

Appendix A Mandatory Standards for File Organization

A.1 Directory Structure

CSSR4Africa software adheres to the standard ROS directory structure for a ROS workspace. There are, in fact, two CSSR4Africa ROS workspaces, one for the physical Pepper robot (`pepper_rob_ws`) and one for the simulator (`pepper_sim_ws`). The directory structure is identical, as shown in Fig. 4 for the `pepper_rob_ws` workspace.

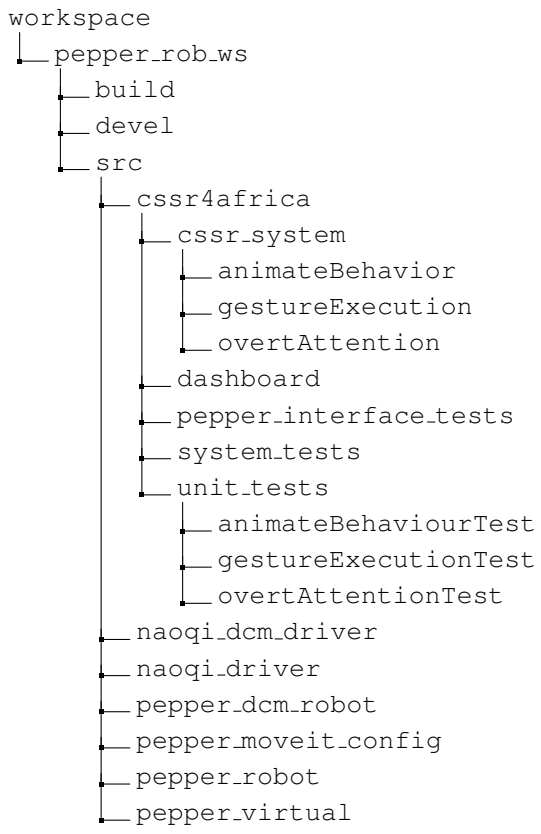

Figure 4: Directory structure for the CSSR4Africa software repository.

The `cssr4africa` directory is a ROS metapackage which collects together the ten ROS packages that make up the CSSR4Africa system. Six of these are CSSR4Africa system architecture packages and four are auxilliary packages, as shown in Table 1.

Each package has a similar directory structure. For example, see the directory structure for `pepper_interface_tests` in Fig. 5. In this case, there are two ROS nodes: `actuatorTest` and `sensorTest`. Eventually, the six system architecture packages will be populated with the components (nodes) specified in the CSSR4Africa work plan, augmented by any other components that are identified when designing the system architecture in Task 3.1.

System Architecture Packages	
Architecture Subsystem	Package Name
Animate Behaviour	`cssr_system`
Attention	`cssr_system`
Gesture, Speech, & Navigation	`cssr_system`
Detection of Interaction Events	`cssr_system`
Interaction Manager	`cssr_system`
Sensing & Analysis	`cssr_system`
Test Packages	
Test Type	Package Name
Sensor and Actuator Tests	`pepper_interface_tests`
System Tests	`system_tests`
Unit Tests	`unit_tests`

Table 1: The ROS packages that comprise the CSSR4Africa software system.

A.2 Filename Roots and Extensions

A.2.1 C/C++Programming Language

All files should have the same root, reflecting computational purpose of the component, e.g., `exampleComponent`. Other filenames are derived from this by appending the appropriate string to denote the files function. For example, an `exampleComponent` node would have the following files. These are explained in the following sections.

- `exampleComponentApplication`.
- `exampleComponentImplementation`.
- `exampleComponentInterface`.
- `exampleComponentConfiguration`.
- `exampleComponentInput`.
- `exampleComponentOutput`.
- `exampleComponentDriver`.
- `exampleComponentStub`.
- `exampleComponentLaunchRobot`.
- `exampleComponentLaunchSimulator`.
- `exampleComponentLaunchTestHarness`.

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.

```
pepper_interface_tests
├── config
│   ├── actuatorTestConfiguration.ini
│   └── sensorTestConfiguration.ini
├── data
│   ├── pepperTopics.dat
│   ├── simulatorTopics.dat
│   ├── sensorTestOutput.dat
│   ├── actuatorTestInput.dat
│   └── sensorTestInput.dat
├── include
│   └── pepper_interface_tests
│       ├── actuatorTestInterface.h
│       └── sensorTestInterface.h
├── launch
│   ├── actuatorTestLaunchRobot.launch
│   ├── sensorTestLaunchRobot.launch
│   └── interfaceTestLaunchSimulator.launch
├── src
│   ├── actuatorTestApplication.cpp
│   ├── actuatorTestImplementation.cpp
│   ├── sensorTestApplication.cpp
│   └── sensorTestImplementation.cpp
├── README.md
├── CMakeLists.txt
└── package.xml
```

Figure 5: Directory structure for the `pepper_interface_tests` package. There are two nodes: `actuatorTest` and `sensorTest`.

A.2.2 Python Programming Language

All files should have the same root, reflecting computational purpose of the component, e.g., `example_component`. Other filenames are derived from this by appending the appropriate string to denote the files function. For example, an `example_component` node would have the following files. These are explained in the following sections.

- `example_component_application`.
- `example_component_implementation`.
- `example_component_interface`.
- `example_component_configuration`.
- `example_component_input`.
- `example_component_output`.
- `example_component_driver`.
- `example_component_stub`.
- `example_component_launch_robot`.

- `example_component_launch_simulator`.
- `example_component_launch_test_harness`.

Source code files for python should use a `.py` extension.

Header files should have a `.h` extension.

A.3 File Organization

A.3.1 Source Files

The preferred practice for software that supports encapsulation and data hiding, as, arguably, all software should, is for there to be three types of source file, as follows.

1. Application files.
2. Implementation files.
3. Interface files.

The basis for this approach is that application developers should not need to know about the implementation of the classes, methods, functions, and data structures that they are using. This allows the implementation to be changed without affecting the application.

Thus, the *application file*, e.g., `exampleComponentApplication.cpp` or `example_component_application.py`, contains the `main()` function and the code that instantiates the classes, methods, functions, and data structures to effect the required functionality for the application in question.

The *implementation file*, e.g., `exampleComponentImplementation.cpp` or `example_component_implementation.py`, contains the definitions of the classes, methods, functions, and data structures that implement the algorithms used in the implementation. Ideally, to facilitate reuse, the implementation should be quite general.

The *interface file*, e.g., `exampleComponentInterface.h`, contains the declarations required to use the classes, methods, functions, and data structures that implement the solution to this problem. Thus, the interface file must furnish all the necessary information to use these classes, methods, functions, and data structures.

The application and implementation files `#include` the `exampleComponentInterface.h` file.

In the particular case of CSSR4Africa software, the application file contains the code that instantiates the ROS node along with any required classes, reads the `.ini` configuration file to set the parameter values that govern the behaviour of the node, and calls the functions (C) or invokes the methods (C++ or Python) that provide the required functionality.

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

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

The application and implementation files should be placed in the `src` directory, and the interface file in the node subdirectory in the `include` directory; see Fig. 5. Source files for any driver or stub nodes associated with the unit test , e.g., `exampleComponentDriver.cpp` or

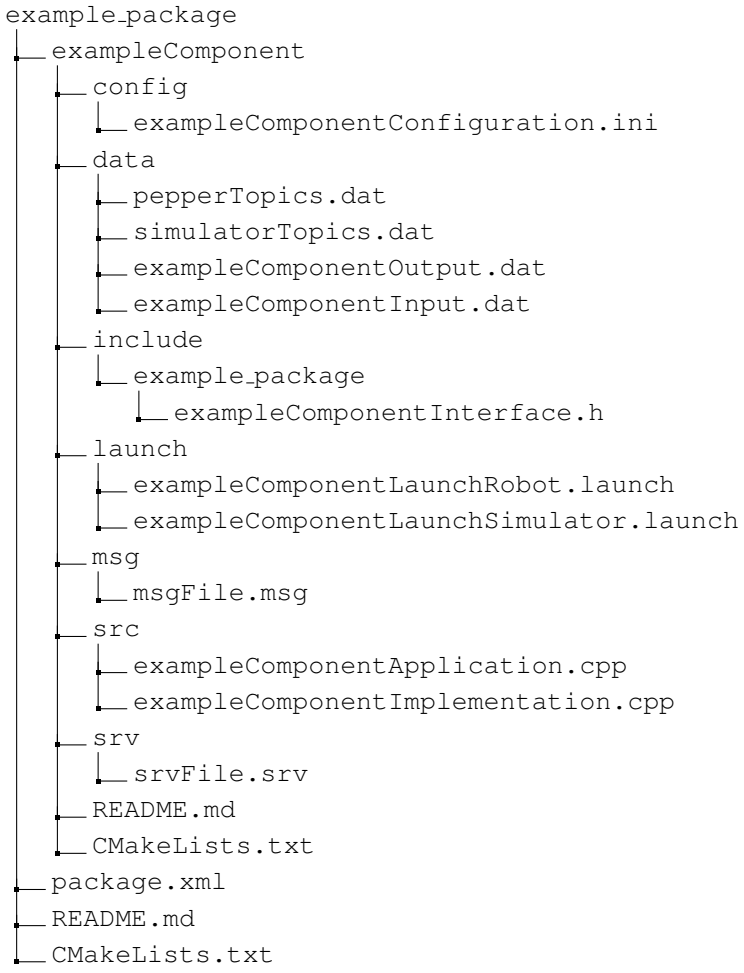

Figure 6: Directory structure for the software written in C++

`example_component_driver.py`, and `exampleComponentStub.cpp` or `example_component_stub.py`, should also be placed in the `src` directory.

Figure 6 shows the directory structure for software written in C++ while Figure 7 the directory structure for software written in Python. It shows the package (`example_package`) which is similar to the `cssr_system` or `unit_tests` package, and the node (`exampleComponent` or `example_component`) which is similar to the nodes specified in Table 1 in Deliverable D3.1 System Architecture.

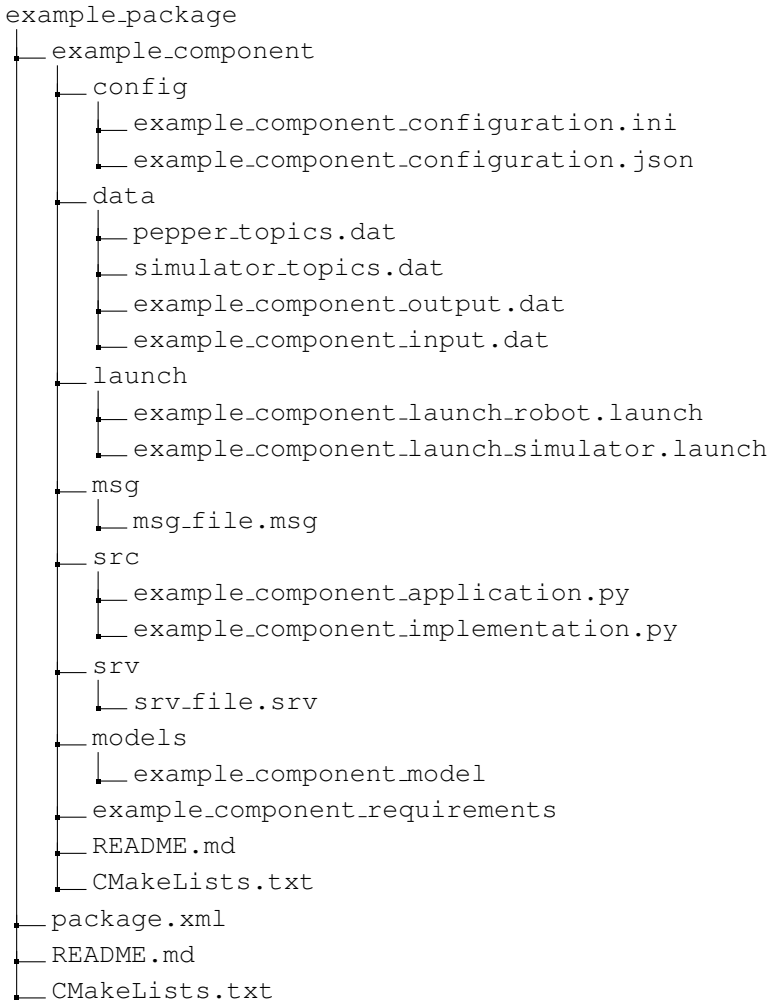

Figure 7: Directory structure for the software written in Python

A.3.2 Launch Files

The `launch` directory should contain up to three launch files, one for use with the physical robot, one for use with the simulator, and one for use with the driver and stub test harness. They should be named after the component but with a `.launch` filename extension. For example `exampleComponentLaunchRobot.launch`, `exampleComponentLaunchSimulator.launch`, and `exampleComponentLaunchTestHarness.launch`, or `example_component_launch_robot.launch`, `example_component_launch_simulator.launch`, and `example_component_launch_test_harness.launch`.

These launch files are in effect the component unit tests and 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, and subsequently transferred to the `unit_tests` package. As such, they should instantiate any necessary driver or stub nodes, the simulator, or the Pepper robot drivers. Instructions on how to run the tests should be included in a `README.md` file in the same directory.

A.3.3 Configuration Files

Each component must have an associated configuration file, named after the component, e.g., `exampleComponentConfiguration.ini` for C++, and `example_component_configuration.ini` for Python. It is placed 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.

A.3.4 Data Files

A component may have an associated input or output data files. In general, these should be named after the component, appended with either `Input` or `Output`, and have either a `.dat` or `.xml` filename extension, e.g., `exampleComponentInput.dat` for C++, and `example_component_input.dat` for Python. However, other filenames are allowable, provided they have a `.dat` or `.xml`= filename extension. These files are placed in the `data` directory.

A.3.5 Other Files

Three other files are stored in the package directory, as follows.

1. `example_component_requirements.txt`
2. `CMakeLists.txt`
3. `README.md`
4. `package.xml`

The `README.md` file provides information about the package in a Markdown markup language. It should describe each node in the package and explain how to use them. The content will be based on the documentation provided with nodes when they are submitted for integration and inclusion in the release of CSSR4Africa repository on GitHub.

The `CMakeLists.txt` file contains the build directives for each node in the package. It will be based on the CMakeLists.txt that are provided with nodes when they are submitted for integration and inclusion in the release of CSSR4Africa repository on GitHub.

The `package.xml` file contains the package *manifest*. This defines details about the package, such as the name, version, maintainer, and dependencies.

The `example_component_requirements.txt` file contains the necessary modules and dependencies required to be installed for software written in Python programming language. Guidelines to install these require modules are specified in the `README.md` file.

Appendix B Mandatory Standards for Internal Source Code Documentation

B.1 General Guidelines

Two types of comments are required: documentation comments and implementation comments. Documentation comments describe the functionality of a component from an implementation-free perspective. Together with the contents of the node `README.md` file, they will be used to create the documentation in the component reference manual on the CSSR4Africa wiki. 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), e.g, the ROS topics, services, and actions it uses to communicate with other ROS nodes, rather than understand its implementation. Implementation comments explain or clarify some aspect of the code. They 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.

Documentation comments are delimited by `/* ... */` in C++ and `""" ... """` in `Python`.

Implementation comments are delimited by `/* ... */` and `//` in C++ and `#` in `Python`.

All comments should be written in English.

B.2 Documentation Comments

The First Documentation Comment
All C++ source files must begin with a documentation comment that identifies the file being documented, and gives a copyright notice, as follows.

```
/* <filename> <one line to identify the nature of the file>
 *
 * Author:
 * Date:
 * Version:
 *
 * Copyright (C) 2023 CSSR4Africa Consortium
 *
 * This project is funded by the African Engineering and Technology Network (Afretec)
 * Inclusive Digital Transformation Research Grant Programme.
 *
 * Website: www.cssr4africa.org
 *
 * This program comes with ABSOLUTELY NO WARRANTY.
 */
```

All Python source files must begin with a documentation comment that identifies the file being documented, and gives a copyright notice, as follows.

```
""" <filename> <one line to identify the nature of the file>

     Author:
     Date:
     Version:

     Copyright (C) 2023 CSSR4Africa Consortium

     This project is funded by the African Engineering and Technology Network (Afretec
     Inclusive Digital Transformation Research Grant Programme.

     Website: www.cssr4africa.org

 This program comes with ABSOLUTELY NO WARRANTY.

 """
```

The <componentName>Application.cpp file must also contain additional information, documenting the component API. The following is a list of the mandatory sections for which documentation comments must be provided.

```
/* <componentName>Application.cpp <one line to identify the nature of the file>
 *
 * <detailed functional description>
 *
 ...
 * Libraries
 ...
 * Parameters
 *
 * Command-line Parameters
 ...
 * Configuration File Parameters
 ...
 * Subscribed Topics and Message Types
 ...
 * Published Topics and Message Types
 ...
 * Input Data Files
 *
 * <componentName>Input.dat
 ...
 * Output Data Files
 *
 * <componentName>Output.dat
 ...
 * Configuration Files
 *
 * <componentName>Configuration.ini
 ...
 * Example Instantiation of the Module
 *
 * rosrun <componentName> __name:=<alternativeComponentName> ...
 ...
 *
 * Author:  <name of author>, <author institute>
 * Email:   <preferred email address>
 * Date:
 * Version:
 *
 */
```

The <component_name>_application.py file must also contain additional information, documenting the component API. The following is a list of the mandatory sections for which documentation comments must be provided.

```
""" <component_name>_application.py <one line to identify the nature of the file>
    <detailed functional description>

 ...
    Libraries
 ...
    Parameters

    Command-line Parameters
 ...
    Configuration File Parameters
 ...
    Subscribed Topics and Message Types
 ...
    Published Topics and Message Types
 ...
    Input Data Files

    <component_name>_input.dat
 ...
    Output Data Files

    <component_name>_output.dat
 ...
    Configuration Files

    <component_name>_configuration.ini
 ...
    Example Instantiation of the Module

    rosrun <component_name>_application.py __name:=<alternative_component_name> ...
 ...

    Author:  <name of author>, <author institute>
    Email:   <preferred email address>
    Date:
    Version:

 """
```

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.

In C++, 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.
 */
```

In, Python, block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

```
#
# Here is a block comment.
#
```

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.

In C++ ,

```
if (condition) {

    /* Handle the condition. */

    ...
}
```

In Python ,

```
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.

In C++,

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

In Python, trailing comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

```
if a == b:
   return True                  # special case

else:
   return general_answer(a)     # only works if a != b
```

End-Of-Line Comments

In C++, 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
//}
```

In Python, 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 Recommended Standards for Programming Style

C.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.

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
```

C.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.

C.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.

C.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:
In C++,

```
return;

return myDisk.size();

return TRUE;
```

In Python,

```
return

return myDisk.size()

return True
```

`if`**,** `if-else`**,** `if else-if else` **Statements**
In C++, 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;
```

In Python, The `if-else` class of statements should have the following form:

```
if condition:
   statements


if condition:
   statements
else:
   statements

if condition:
   statements
elif condition:
   statements
else:
   statements
```

`for` Statements

In C++, `for` statement should have the following form:

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

In Python, `for` statement should have the following form:

```
for variable in collection:
   statements
```

`while` Statements

In C++, `while` statements should have the following form:

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

In Python, `while` statements 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.

C.5 Naming Conventions

C vs. C++ vs. Python
Naming conventions make programs more understandable by making them easier to read. Since CSSR4Africa software uses the C language, the C++ language, and the Python language, sometimes using the imperative programming and object-oriented programming paradigms separately, sometimes using them together, we will adopt three different naming conventions, one for C, one for C++, and the other for Python. The naming conventions for C++ are derived from the JavaDoc standards [23], while the naming conventions for Python are derived from the Python Enhancement Proposals standards [26].

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 _	`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))`

Python 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 all lowercase with words separated by underscores _	`def grab_image():` `def set_velocity():`

Variables	variable names should be all lowercase with words separated by underscores _	`i`
		`f`
		`pixel_value`

Constants	The names of variables declared as constants should be all uppercase with words separated by underscores _	`WIDTH = 4`

C.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 C++. 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 D Recommended Standards for Programming Practice

D.1 C++ and Python 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.

D.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 C.1:

```
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.

D.3 General Issues

Use of Guards for Header Files in C++
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 in C++
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 in C++
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.

In C++,

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

should be written as

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

In Python,

```
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 [27] satisfies this requirement.

Error Messages
Error messages should look like this:

```
function_name: error message
```

Copyright Messages
The program, i.e., the ROS node, should write to the terminal the following short notice when it starts.

```
<Program name and version>

This project is funded by the African Engineering and Technology Network (Afretec)
Inclusive Digital Transformation Research Grant Programme.

Website: www.cssr4africa.org

This program comes with ABSOLUTELY NO WARRANTY.
```

Start-up Messages
The program, i.e., the ROS node, should write short messages to the terminal during the start-up phase to indicate the state of the node. They should look like this:

```
nodeName: start-up.
nodeName: subscribed to /topicName.
```

Heartbeat Messages
The program, i.e., the ROS node, should periodically (e.g., every five seconds) write a short heartbeat message to the terminal to indicate the state of the node. It should look like this:

```
nodeName: running.
```

To write messages, use `ROS_INFO` for C and `ROS_INFO_STREAM` for C++.

References

[1] http://www.dream2020.eu.

[2] D. Vernon. Software engineering standards. DREAM Deliverable D3.2, 2014. http://www.dream2020.eu.

[3] http://www.robotcub.eu.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Robot Operating System (ROS). `https://www.ros.org/`.

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

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

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

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

[26] *PEP 8 – Style Guide for Python Code*. https://peps.python.org/pep-0008/.

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

Principal Contributors

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

Adedayo Akinade, Carnegie Mellon University Africa.
David Vernon, Carnegie Mellon University Africa.

Document History

Version 1.0
First draft.
David Vernon.
26 October 2023.

Version 1.1
Corrected due date.
David Vernon.
27 October 2023.

Version 1.2
Fixed an unresolved reference to Section C.1 in Section D.2: C Language Conventions.
David Vernon.
30 October 2023.

Version 1.3
Changes to Appendix A (updated Fig. 5 and increased the number of required launch files) and Appendix B (consolidated first block comment and first documentation comment).
David Vernon.
1 November 2023.

Version 1.4
Rationalized the names of the launch files for the pepper_interface_tests package, combining the actuator and sensor versions for the simulator into one file: interfaceTestLaunchSimulator.launch.
Fixed a typo in actuatorTestInterface.h.
David Vernon.
28 November 2023.

Version 1.5
Changed incorrect node name `actuationTest` to `actuatorTest`.
David Vernon.
29 December 2023.

Version 1.6
Simplified ROS package structure for the system architecture subsystems by consolidating all the constituent nodes into one package `cssr_system`, rather than having a separate package for each subsystem.
David Vernon.
30 December 2023.

Version 1.7

Updated directory structure in Fig. 5, moving `actuatorTestInput.ini` to the `data` subdirectory and renaming it `actuatorTestInput.dat`; same for `sensorTestInput.ini`
Changed README.txt to README.md in Section 4.1.
Added `.xml` extensions for data files and relaxed the requirement that the filename had to be `<module_name>Input` or `<module_name>Output`.
David Vernon.
25 January 2023.

Version 1.8

Updated the software standards to accomodate for Python programming language, while distinguishing between C++ and Python in appropriate sections.
Added documentation for application and implementation python source codes named `exampleComponentApplication.py` and `exampleComponentImplementation.py` respectively in Appendix A.3.1.
Added guidelines for documentation and implementation comments in Appendix B.2.
Added guidelines for statements (`if-else`, `return`, `for`, and `while` statements) in Python in Appendix C.4.
Added guidelines for naming conventions in Python in Appendix C.5.
Added directory structure for software written in C++ (Figure 6) and Python (Fgure 7).
Adedayo Akinade.
29 August 2024.

Version 1.9

Updated the software standards to accomodate standard practice for filenames when using the Python programming language, i.e., using an underscore _ to separate words, rather than using camel case, as is the recommended practice with C++.
Thus, the filenames in the version 1.8 entry above now read `example_component_application.py` and `example_component_implementation.py`.
The same practice is also applied to related files, e.g., input, output, and configuration files.
Added guidelines for runtime start-up and heartbeat messages to Appendix D, and modified the copyright message slightly.
Added Adedayo Akinade to the list of contributors.
David Vernon.
9 September 2024.

Version 1.10

Corrected several errors and omissions in the changes made in the previous version.
David Vernon.
13 September 2024.

Version 1.11

Corrected the file directory structure based on updates made in D3.4 System Integration and Quality Assurance Manual.
Added a description for the requirements file needed for software written in Python programming language in A.3.5.
Adedayo Akinade.
3 December 2024.

Version 1.12

Added a `utilities` sub-directory that contains stand-alone software, such as the C++ helper classes that provide access to the culture knowledge base and the environment knowledge base. These classes are used by the ROS nodes that need to access this knowledge; see Deliverable D3.1 System Architecture.
David Vernon
17 February 2025.

Version 1.13

Removed the `utilities` sub-directory since the C++ helper classes will be integrated directly in the software for the `behaviorController` node and not be made available independently.
Added a `dashboard` sub-directory for the dashboard visualization software.
Added `animateBehavior`, `gestureExecution`, and `overtAttention` sub-directories to the `cssr_system` directory.
David Vernon
11 March 2025.

Version 1.14

Changed version of Ubuntu from 18.04 to 20.04 in Section 2.
David Vernon
14 April 2025.