D3.7 – TEST AND INTEGRATION PLAN

Deliverable ID	**D3.7**
Deliverable Title	**Test and Integration Plan**
Work Package	**WP3 – Architecture design and Component Integration**

Dissemination Level	**PUBLIC**

Version	**1.0**
Date	**2017-10-03**
Status	**Final**

Lead Editor	**FRAUNHOFER**
Main Contributors	**Junhong Liang (FRAUNHOFER), Farshid Tavakolizadeh (FRAUNHOFER)**

Published by the CPSwarm Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731946.

Document History

Version	Date	Author(s)	Description
0.01	2017-09-10	Junhong Liang (FRAUNHOFER)	Initial TOC
0.02	2017-09-15	Junhong Liang (FRAUNHOFER)	Added initial content to the document
0.03	2017-09-21	Farshid Tavakolizadeh (FRAUNHOFER)	Modified and enriched content to the document
0.1	2017-09-22	Junhong Liang (FRAUNHOFER)	Released for initial review
0.11	2017-09-28	Junhong Liang (FRAUNHOFER)	Applied comments from the first internal review
0.12	2017-09-29	Farshid Tavakolizadeh (FRAUNHOFER)	Applied comments from the first internal review
0.20	2017-09-29	Junhong Liang (FRAUNHOFER)	Released for second review
0.21	2017-10-02	Farshid Tavakolizadeh (FRAUNHOFER)	Applied comments from the second internal review
0.22	2017-10-02	Farshid Tavakolizadeh (FRAUNHOFER)	Minor modifications, released for approval
1.0	2017-10-03	Farshid Tavakolizadeh (FRAUNHOFER)	Ready for submission

Internal Review History

Review Date	Reviewer	Summary of Comments
2017-09-27	Edin Arnautovic (TTTech)	Accepted with comments regarding the clarity of several sections
2017-10-02	Miquel Cantero (ROBOTNIK)	Comments regarding the clarity of several sections, proofreading
2017-10-03	Miquel Cantero (ROBOTNIK)	Accepted

Table of Contents

1 Executive Summary

This document is a deliverable of the CPSwarm project (Project ID: 731946) funded by the European Commission's Directorate-General for Research and Innovation (DG RTD), under the Horizon 2020 Research and innovation Program (H2020). It is a public report that describes the test and integration plan for software development within CPSwarm.

This document presents the test and integration plan for software components of the CPSwarm project. First, the document provides fundamental concepts, principles, and standards for software testing and continuous integration. Furthermore, the document offers an outline of the strategy and guidelines on how to apply testing and integration concepts, enabling iterative and distributed development of CPSwarm workbench components. Lastly, a software integration timeline in accordance with CPSwarm's Description of Action is illustrated to be used as reference timeframe for development tasks. These guidelines shall be followed by all partners during all future development activities.

2 Introduction

This document describes the test and integration plan for software development in CPSwarm. Note that this document focuses on the testing of software components (e.g. if the components function correctly and fulfil the integration requirements). The testing of algorithms deployed on target hardware (e.g. drones) and their behaviour will be considered in validation tests as part of the WP8 and described in corresponding deliverables.

In the first part of the document, some basic concepts of software testing as well as integration are introduced. For software testing, they include the following relevant topics:

- **Software quality model:** This section discusses software quality models that provide the guidelines for examining the quality of software.
- **Testing approach:** This section provides a guideline for deciding which aspects of a software should be tested and how they should be tested.
- **Testing levels:** This section explains different testing levels, involving unit testing, integration testing, and system testing, each of which helps the developers to build high-quality software.

For software integration, the following topics are illustrated:

- **Source code management:** This section explains the basic practice of source code management in modern software development
- **Continuous integration:** This section illustrates the principles, process, and issues regarding continuous integration, as CPSwarm adopts this strategy for software integration.

The second part of the document presents the test and integration plan of CPSwarm. In this part, a test plan for CPSwarm regarding the above-mentioned principles is outlined and then strategy and responsibilities for different testing levels are defined as guidelines for future development activities. Besides, tools that will be used for code management (Gitlab) as well as continuous integration (Atlassian Bamboo) in CPSwarm are introduced with a demonstration of the setup. In the end, a timeline for the integration is presented according to CPSwarm's Description of Action.

2.1 Related documents

ID	Title	Reference	Version	Date
D2.1	Initial Vision Scenarios and Use Case Definition	D2.1	2.0	31.05.2017
D3.1	Initial System Architecture Analysis and Design Specifications	D3.1	1.0	31.06.2017

3 Software Testing

3.1 Software Quality Models

Software testing is an important way to ensure software quality. The term software quality is an abstract concept and its meaning may vary depending on the context. Therefore, a comprehensive software quality model for the early evaluation of software quality is needed. There are many models of software product quality that define software quality attributes.

Three often used models are discussed as examples:

- **McCall's model** of software quality (McCall, Richards, & Walters, 1977) incorporates 11 criteria encompassing product operation, product revision, and product transition
- **Boehm's model** (Boehm, 1989) is based on a wider range of characteristics and incorporates 19 criteria. Criteria in these models are not independent; they interact with each other and often cause conflict, especially when software providers try to incorporate them into the software development process
- **ISO/IEC 25010 model** (ISO/IEC., 2011) incorporates eight quality characteristics, each having a large number of attributes

The criteria and goals defined in each of these models are listed in Table 1. Note that the ISO Model groups a number of criteria under its characteristics of maintainability.

Table 1 - Software Quality Models

Criteria/Characteristics	McCall	Boehm	ISO/IEC 25010
Correctness	X	X	maintainability
Reliability	X	X	X
Integrity	X	X	
Usability	X	X	X
Efficiency	X	X	X
Maintainability	X	X	X
Testability	X		maintainability
Interoperability	X		
Flexibility	X	X	
Reusability	X	X	
Portability	X	X	X
Clarity		X	
Modifiability		X	maintainability
Documentation		X	
Resilience			
Understandability		X	
Validity		X	maintainability
Functionality			X

Generality		X	
Economy		X	

Out of these models, the ISO 25010 model is the most recent and comprehensive one and will therefore be the guideline for quality assurance and validation process. However, as a research project, a complete compliance to the model is out of scope. As a result, the most important characteristics such as functional suitability, reliability, operability will be the primary focus in quality control. To achieve that, we will make best efforts for incorporating these characteristics into various development stages in CPSwarm.

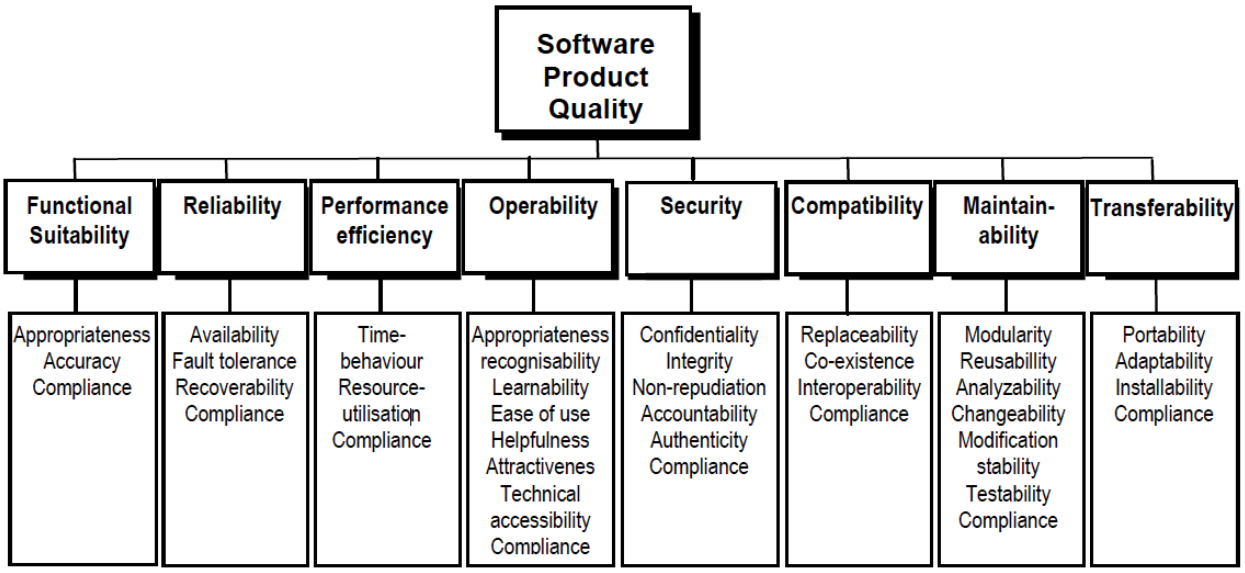
Software Product Quality
Functional Suitability
Reliability
Performance efficiency
Operability
Security
Compatibility
Maintain-ability
Transferability
Appropriateness Accuracy Compliance
Availability Fault tolerance Recoverability Compliance
Time-behaviour Resource-utilisation Compliance
Appropriateness recognisability Learnability Ease of use Helpfulness Attractivenes Technical accessibility Compliance
Confidentiality Integrity Non-repudiation Accountability Authenticity Compliance
Replaceability Co-existence Interoperability Compliance
Modularity Reusabillity Analyzability Changeability Modification stability Testability Compliance
Portability Adaptability Installability Compliance

Figure 1 ISO/IEC 25010 Quality model for external and internal quality

The standard distinguishes three gradually dependent dimensions of the software product quality:

1. Internal quality refers to the "internal", developer view on static aspects of the system, i.e. the architecture and implementation of the system.
2. External quality refers to the externally observed behaviour and functioning of the system while being executed and in a simulated/controlled environment ("developers' environment").
3. Quality in use refers to user's perception of the system quality while achieving goals in a particular environment/context of use.

The internal and external quality is modelled in terms of 8 characteristics and numerous sub-characteristics as depicted in Figure 1. A short description of each characteristic is given in Table 2. These characteristics provide a valuable, high-level model for quality assessment of a software system; the models will be covered throughout the various stages of the CPSwarm testing and quality assurance process.

Table 2 - Quality characteristics of a software system in ISO/IEC 25010

Quality Characteristics	Description
Functional Suitability	Functional coverage that meets stated and implied needs when the software is used under specified conditions.
Reliability	Maintenance of a specified level of performance when used under specified conditions.

Operability	Capability of the system to be comprehensible, operable and attractive to the user, when used under specified conditions
Performance Efficiency	Appropriate performance, relative to the amount of resources used, under stated conditions
Security	Proper protection of assets in their various aspects
Compatibility	Aspect of the system to be compliant to standards, providing interoperable interfaces, and being replaceable with other standard compliant components
Maintainability	Capability of the software product to be modified (corrected, improved or adapted to changes in environment and specifications)
Transferability	Capability of the system to be transferred from one environment (development) to another (production)

Complementary to considering the quality of system itself, the quality in use refers to system's effective usage and perception by the end user as shown in Table 3. It refers to the overall capability of the system to enable specified users to achieve goals with regards to its quality characteristics: effectiveness, productivity, safety, satisfaction, and usability. The quality in use conveys user's perspective on the system quality. It is measured by results of its concrete usage in context, rather than by properties of the software itself, and its evaluation will therefore involve participation of users of the CPSwarm platform. This document considers the quality in use of software components of the CPSwarm workbench (for modelling, optimization, deployment, and monitoring).

Table 3 Quality in use characteristics of a software system

Quality Characteristic	Description
Effectiveness	System's capability to enable users to accurately and completely achieve specified goals in a particular context of use
Productivity	System's capability to enable users to expend appropriate amounts of resources (time, effort, money etc.) in relation to the effectiveness achieved in a specified context of use
Safety	System's capability to limit risk of harm to people, business, environment etc. to an acceptable level
Satisfaction	System's capability to satisfy users in a specified context of use
Usability	The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use

3.2 Testing Approaches

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test (Kaner, 2006). In general, there are numerous dimensions of a testing approach and a continuum of values within each dimension (McGregor & Sykes, 2001):

- When will testing be performed? Will testing be done every day, as components are developed, or when all components are put together?
- Who performs the testing? Are developers responsible or are independent testers responsible?

- Which pieces will be tested? Will everything, a sample, or nothing be tested?
- How will testing be performed? Will testers have knowledge of only the specification of the component under test (black-box testing) or also knowledge of implementation (white-box testing)?
- How much testing is adequate? Will no testing be done or will exhaustive testing be done?

The choice of approach is dependent on the chosen development approach. The development approach in CPSwarm is iterative and incremental, comprising several complete cycles of analysis, development, and validation in which components from different partners are frequently integrated.

It is increasingly clear that the approach to testing in such conditions should preferably be "agile" or "lean" (Beck, 2000). This means that CPSwarm should use automatic testing whenever possible. In addition, "adequate" testing needs to be seen in the context of what is currently needed of the platform. For demonstrator applications, the primary objective is to demonstrate partial functionality. Thus, focus is here not on doing a complete acceptance test (or even tests conforming to statement coverage), but rather on integration testing.

3.3 Testing Levels

Testing is typically considered to take place on four different levels (Beizer, 1990):

1. Unit testing – Testing at the lowest level using a coverage tool. A unit is the smallest testable part of an application
2. Integration testing – Testing of interfaces between components to ensure that they are compatible
3. System testing – Testing of entire software systems
4. Validation – Evaluation at the end of development to ensure requirements fulfilment.

Software should first be unit tested, followed by integration and system tests. As CPSwarm has an iterative approach, there might be interleaving or backtracking of these testing activities. These tests are done internally by the CPSwarm system developers. On the other hand, validation tests to examine all the integrated business scenarios and defined KPIs will be performed based on the demonstrator applications as developed in WP8 and is not further described in this document.

3.3.1 Unit Testing

A "unit" is the smallest testable part of an application, a concise atomic unit of function. Unit tests are commonly automated and written using a unit testing framework specific to programming language environment.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. It offers several advantages for software developments, such as early error detection and localization, uncertainty reduction, etc. Unit testing can help to simplify the difficulties and reduce the efforts for later integration tests.

In practice, unit tests are carried out with the following basic guidelines (in context of object-oriented programming):

- For each class, there should be a test class which tests the public methods.
- Tests cover positive tests, negative tests and limit tests.
- Dependencies to other classes should be substitute by mock objects.
- Each test case covers exactly one functionality to achieve a quick bug fixing.

- For each abstract class, an abstract test class will be implemented. This abstract test class tests the implementation parts of the abstract class and outlines the correct use of the abstract class and the test classes to implement for the concrete classes.

Since they ensure unit's correct behaviour, therefore only components that passed all unit tests may be committed to the source code repository.

Upon completion of an increment of a unit, the following must be considered:

- Code checked into the source code repository must not break the build process
- Prior to committing new versions of a unit to the source code repository there must be reasonable automated, functional unit tests. There will be no required test coverage criteria, but we advise a test-coverage of 50 percent or higher

To enhance quality of the software it is recommended to create a new unit test for each detected bug if there was not one already. After fixing the bug the commit statement has to contain the bug number so that an automated tracking of bug fixes can be performed.

3.3.2 Integration Testing

Integration testing takes place whenever multiple units need to work together. It focuses on the correct interaction of components in order to provide a higher-level functionality. Thus, it considers merely runtime factors like compatibility of their (local or remote) interfaces, service and dependency resolution, and quality of service (QoS) contracts between the components (timeouts, error indication etc.).

In the CPSwarm context this is a test over several components in the CPSwarm system. Integration tests are grey box tests. That means that a complete workflow will be tested and the results will be checked. The workflow structure and implementation is partially known and considered.

The CPSwarm system will be developed in different environments as well as on different platforms. The tools and frameworks for integration testing will be specific to programming and development environments.

3.3.3 System Testing

While the integration testing focuses on interoperability between components, the main purpose of System testing is to verify in a practical way that all components function correctly together and the overall CPSwarm system includes all the technical capabilities needed to support the foreseen scenarios.

For this reason, system test can be considered as a step to assess the minimum required CPSwarm technical capabilities, and this is needed before CPSwarm evaluation activities can take place.

System testing will be performed after integration testing, and will be performed in a horizontal manner assuming that integration will take care of the vertical interoperability across the different levels of the system. These tests will be automated whenever possible. In case of manual tests, the tester runs a defined test protocol against the system and checks the functional requirements and the non-functional requirements as well.

4 Software Integration

This chapter describes the fundamental software integration concepts. In CPSwarm, the software integration involves source code management as well as continuous integration. The principles and practices in these domains will be elaborated in the following sections.

4.1 Source Code Management

The envisioned CPSwarm workbench consists of several independent and loosely coupled components. Most components are under active development by project partners, the open-source communities, and other organizations. In such a versatile environment, it is important to utilize the right version control and issue management tools to enable smooth distributed development by different parties.

4.2 Continuous Integration

As mentioned earlier, CPSwarm utilizes an agile and distributed development, which makes the integration of different components a central concern throughout the whole development phase. Generally, the following different integration strategies are possible for software development:

Big Bang Integration
In principle, it is possible to integrate all (or many) components in one single step. This approach is also called "big-bang-integration". Test drivers and placeholders are unnecessary because the integration is performed without any additional aid when the system is complete. Nevertheless, integration in one step is marginally considered in practice. If each component exhibits errors with a certain probability and inconsistencies introduce additional errors, the testers need to work with a system that is barely executable. Uncovering errors becomes complex, since these errors are spread across a large system, which may not be known to the testers. Even if the developers themselves participate in the testing, they generally only possess detailed knowledge of their own components. Thus, most software development approaches prefer an incremental integration described below.

Incremental Integration
The incremental integration is performed on a case-by-case basis. Three variants of incremental integration exist:

- **Top-Down Integration:** The top-down integration follows the hierarchical structure of the architecture. If the architecture defines a layered structure, the main class and the components hosted by the upper layer are integrated first. Thereafter, the components on the next underlying layer are integrated, and so forth. The advantage of this strategy lies in the fast availability of an executable system. On the other hand, the disadvantage is that potentially many placeholders need to be constructed. In order to make the system truly executable, these place holders need to offer a certain level of functionality and accordingly their construction is complex.

- **Bottom-Up Integration:** In this approach, testing is conducted from sub module to main module, if the main module is not developed a temporary program called DRIVERS is used to simulate the main module. With this approach, fault is more easily located compared to big bang integration and no time is wasted waiting for all components to be available. However, this approach also suffers from disadvantages, like critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects, and an early prototype is not possible.

- **Continuous Integration (CI):** A different approach arises from the Extreme Programming paradigm. In order to rapidly build an executable system and to enable the developers to equally work together on different parts of the system, new and modified components need to be continuously integrated in a simple and fast manner. As soon as a new component is complete, it is integrated and tested. The integration takes place within a separate environment – on a dedicated integration machine – which is detached from the development environment.

The continuous integration approach implies frequent integration and testing of work results. Only if the test does not reveal any errors, the new code remains on the integration machine. Otherwise, the code must be removed and the prior state of the system is reconstructed.

This type of integration requires a close cooperation of all developers. Because the integration happens frequently (e.g. once a day or each time something is checked into the repository). The individual extensions and modifications need to be small. Otherwise, each single integration step takes too long and the integration becomes a bottleneck. The short work cycles require use of automated methods for source code management, configuration and testing.

Due to the diversity of code-base in CPSwarm, it is especially important to avoid Big Bang Integration and practice incremental approaches. In contrast to the traditional 'Big Bang Integration', where software modules are developed in isolation and integrated at the end of the project, Continuous Integration (CI) allows an iterative integration from the beginning of a project, bringing in the major advantage that allows the teams to detect the problems early. Apart from this, CI tools provide detailed reports about the build process and test results, enabling developers to ensure component integration before software deployment and release. Thus, we utilize the continuous integration strategy for component integration and setup a CI environment utilizing an appropriate set of tools. Components that reach a functional state will be added to the integration environment. Consequently, every committed change in the code will trigger automatic builds and integration tests.

However, as long as the system is not completely integrated, the result of each integration step is denoted as "partially integrated system". In general, a partially integrated system is not executable, and therefore, an integration test requires test drivers and placeholder (stubs). A test driver provides the partially integrated system with test input, while a placeholder acts in the place of a component that is required by the partially integrated system. Therefore, extra effort is necessary to develop these placeholders for missing components.

It is also important to note that the current integration plan is based on the software architecture of CPSwarm as described in document D3.1. The integration plan will evolve along with the software architecture, however the overall strategy will remain the same.

4.2.1 Continuous Integration Process

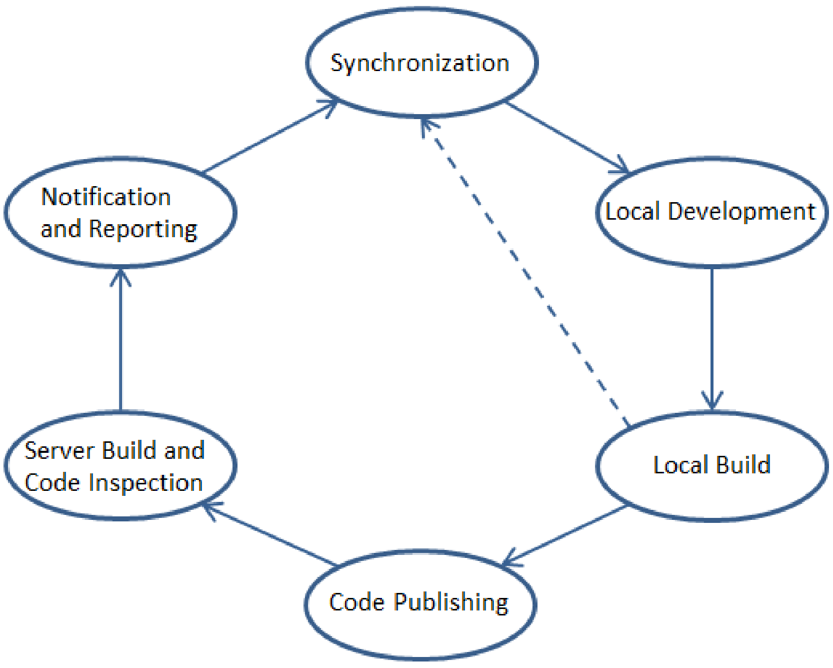

Figure 2 - Continuous Integration Process

A well-designed continuous integration process encapsulates other sub-practices such as continuous builds, continuous unit testing, continuous deployment, continuous notifications, and continuous reporting. All of these are designed to shorten the time between the problem discovery and the problem solution.
The CPSwarm integration process forms a continuous cycle; see Figure 2. The steps of the integration cycles are described as follows:

Synchronization
The first step of the continuous integration process constitutes in the synchronisation of the files on the local developer machine. The developer is checking out a working copy of the source code repository from the mainline (or trunk) of the revision control system. Thus, the developer obtains a copy of the currently integrated source onto his local development machine. This local copy is branched, and can now be modified so that it implements the feature required by the overall software system.

Local Development
During this step, the developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity. The refactoring of their source code often means modifying without changing its external behaviour; the developers are cleaning up their source code. In case that there are already local changes in a file which the developer wants to keep, but there are also changes in repository, the developer has to merge the changes.

Local Build
The developer who works on a software component and changes the system makes every effort to ensure that the new feature does not introduce a bug into the overall system: once this has been done (and usually at various points while during the development) the developer triggers an automated build on his development machine. This takes the currently developed source code, compiles and links it into an executable program, and runs automated or manual tests. These tests may comprise functional, security, load

and performance tests. Only if the entire code builds correctly and runs the tests without errors the overall build is considered to be valid. The unit can be functionally accepted.

Code Publishing
After the local build succeeded the developer can publish the local changes of the source code. This is done by committing the local sources to the central software revision system. In case there are already committed changes to the source code in the software revision system, the source code has to be synchronized again before it can be published.

Server Build and Code Inspection
Whenever a new revision of a software unit is made available and committed to the repository, the automated continuous integration system is notified. The continuous integration server performs a check-out of the most recent version of the software system from the repository. The code is built on the integration server and the automated tests are executed.

With certain plugins, the continuous integration server can also perform static or dynamic analysis checks that address the test coverage of the source code, possible duplication of code and dependencies among classes.

Notification and Reporting
In case the build or unit testing fails, the development team will automatically be notified by the continuous integration server. This notification will be done via email. This ensures that errors are reported back before the programmer forgets what exactly he has done. It also serves to prevent sloppy code check-ins as mistakes will immediately become visible not only to the responsible developer.

Additionally, the continuous integration server can generate tests reports and reports about source code quality.

4.2.2 Fundamental Integration Principles

The following rules can help to avoid common errors and weaknesses of the integration:

1. **Plan the integration:**
   - Only if the integration is planned explicitly, a reasonable and realisable sequence of integration steps arises.
   - An integration plan is a prerequisite for building the system with as small effort as possible.

2. **Integrate early and often:**
   - One approach of integration constitutes in launching the integration before the implementation started.
   - The designed components are realised as placeholders that only match the design with regard to interfaces, and during the implementation they are filled with code.
   - In this way, an executable system is achieved at an early stage.

3. **Capture the total effort of the integration precisely:**
   - A shallow or non-existent integration plan leads to massive problems that put a burden on the budget and schedule of the entire project.
   - Even if an integration plan exists, the planned resources and time frame is underestimated in many cases.

4. **Coordinate the integration with the project organization and development process:**
    - Particularly large systems that are developed in a distributed manner require the architecture to consider the organization of the development process.
    - This in turn influences the integration strategy.
    - The planning of the integration involves the consideration whether the system is developed incrementally or as a whole.

5. **Identify and Reduce the Risks of the Integration:**
    - In particular, the integration of third party components requires a plan of how to deal with low-quality delivery or delivery behind schedule.
    - Possibly other, more elementary components can be integrated instead, or only a leaner version of the system is completed at first.
    - The approach to such problems needs to be prepared in advance.

4.2.3 Problems with Integration

In an ideal world, integration is a simple task. The developers realise their components independently from each other. Since they precisely follow the specification and the syntactical level fits through strict type checking, all parts of the system fit together as planned. The efforts of integration is only marginal. In practice, three reasons are opposed to a trouble-free integration:

- Usually the specification is incomplete and imprecise. Thus, the specification is interpreted differently and sometimes misunderstood by the developers. The result is a set of components that do not fit together.
- Usually a software system includes third party components. In many cases information about these components is entirely insufficient. In addition, such components are not available on time.
- Different hardware sensors will be deployed which constitute specific third party software. This places further demands on the integration process.

Only after the integration, inconsistencies, and contradictions become visible. If the integration is performed at a late point in time, the analysis and correction of defects or the substitution of corrupt components might be expensive and happens in a "panic mode". Modern programming languages and methods allow for a minimisation of such problems. Strict type checking guarantees the syntactic compatibility of the components, but semantic compatibility can only be achieved through accurate planning and specification of the parts before they are implemented.

5 CPSwarm Test and Integration Plan

This chapter presents the test and integration plan in the context of CPSwarm. The plan is made with the previous mentioned principles and standards in mind to ensure best practices. Section 5.1 demonstrates the testing plan, while 5.2 describes the integration plan.

5.1 Testing Plan

In Chapter 3, we elaborated the testing strategy and different testing levels. As part of the continuous integration, we perform three testing levels: unit tests, integration tests, and system tests (see Figure 3).

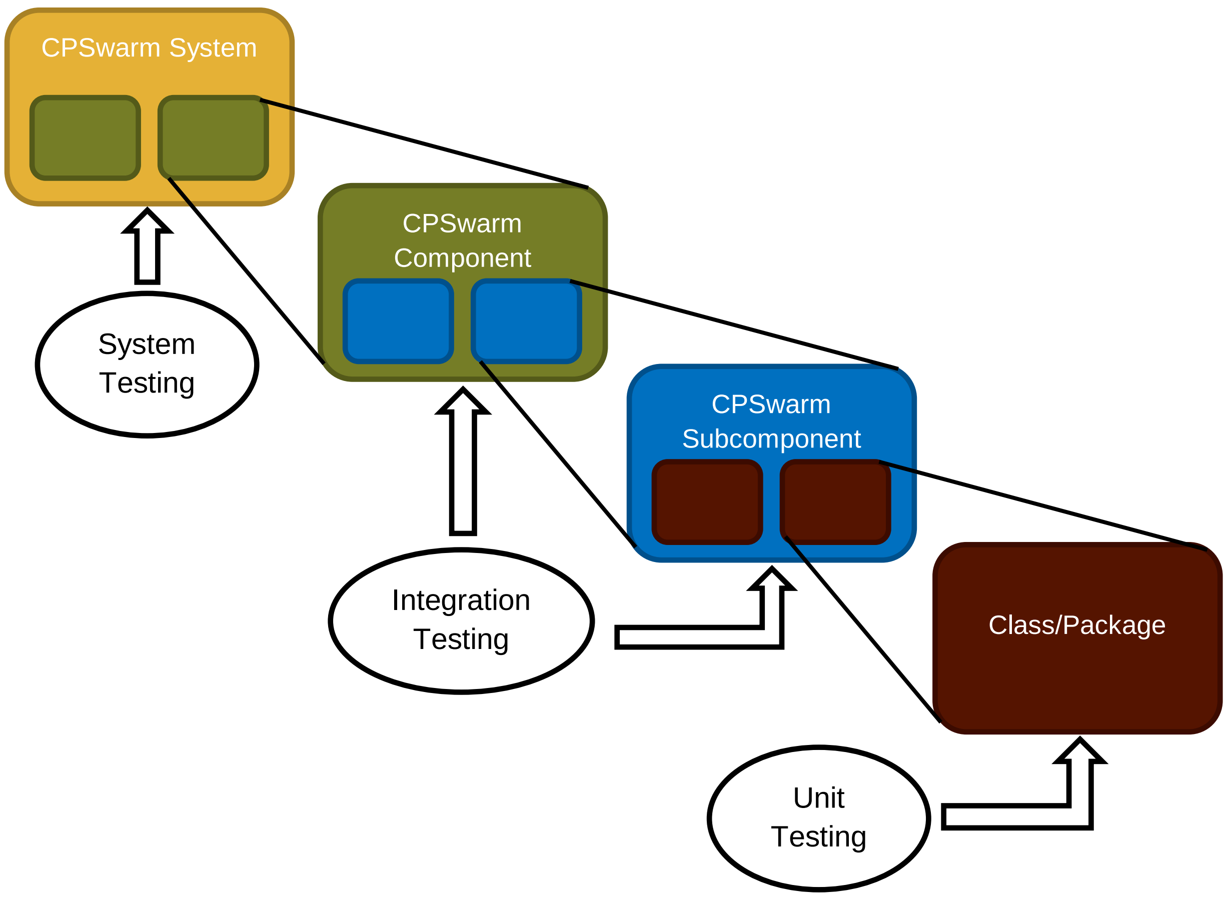

Figure 3 - CPSwarm testing eco-system

Unit testing

The main focus of unit testing is to ensure the correct implementation of the smallest testable units, i.e. classes. In CPSwarm, unit testing is responsibility of each component's developer. However, according to the guidelines in section 3.3.1, there are a few general principles that should be followed by all developing partners within the project:

- For each class, there should be a test class which tests all the public methods.
- Tests cover at least positive tests and negative tests.
- Dependencies to other classes should be substituted by mock objects.
- Each test case covers exactly one functionality to achieve a quick bug fixing.

- For each abstract class, an abstract test class will be implemented. This abstract test class tests the implementation parts of the abstract class and outlines the correct use of the abstract class and the test classes to implement for the concrete classes.

For example, if a class responsible for exporting configuration from Modelling Tool to Optimization Tool exists, all public methods of this class should be tested. Other classes which this class depends on, such as a configuration manager class, should be substituted with a mock object so that the class in test can be tested in isolation. Positive and negative test cases should be written for each public method. In context of this class, test cases should be designed to check if the exported configuration is correct, given the correct input (positive test) and if the system could handle error gracefully, given the faulty input (negative test).

There are many existing tools available to help users performing unit testing on different programming environments. In CPSwarm, proper unit testing tools, such as JUnit for Java, will be chosen according to our development to assist unit testing.

Integration testing

The integration testing of CPSwarm components will be performed in two stages, first between sub-components of a component and then between components. A component is typically composed of a set of sub-components and these sub-components consists of classes and packages.

- **Sub-component integration testing:** The integration tests between sub-components ensure compatibility of these software units across development cycles. As example of sub-component integration test in Design Environment is the validation of outputs from Modelling Library against the expected inputs of the Modelling Tool. It is generally the responsibility of developers to write appropriate test scripts to ensure sub-component integration.

- **Component integration testing:** The integration tests between components will target interoperability of CPSwarm components. These tests will examine the compatibility of components' APIs according to a predefined set of rules. Furthermore, they examine basic functionalities of a group of components as a sub-system. An example of such tests is between Design and Algorithm Optimization environment, where the outputs of the Modelling Tool will be tested against expected inputs of Optimization Tool. FRAUNHOFER who leads integration tasks, will work closely with component developers to create and monitor integration tests between components.

System testing

The system testing runs tests scripts designed for validating the functionality of the system as a whole or as a partially integrated system. For example, a set of tests could validate that a known model results in an expectedly optimized algorithm and could be deployed on a target architecture.

Concretely, the system tests focus on evaluating scenarios described in deliverable document D2.1 related to swarm of drones, automotive CPS, and logistics assistant. This includes testing of the interoperability and functionality of components dedicated to modelling, optimization, deployment, and monitoring. We do not deploy components on the physical runtime environment (e.g. drones) but we perform tests to ensure that codes can be generated for and deployed on a target platform. Furthermore, we mock swarm members who publish data in tests related to the monitoring tool.

Similar to the component integration testing, FRAUNHOFER will lead the task and will work together with component developers to prepare elaborate testing procedures.

The testing activities, testing frequency and responsibility are summarized in Table 4:

Table 4 - Test activities, frequency, and responsible partners for each level of testing in CPSwarm

Level of testing	Activities	Frequency of testing	Responsible Partner(s)
Unit	• Select test cases • Write scripted test cases	• Test creation while developing component • Tests run continuously when component is built on the CI Server	• Responsible partners for the component in test
Integration	• Select test cases • Write scripted test cases • Prepare manual test cases	• Scripted tests run continuously when components are built on the CI Server • Manual tests, at least each iteration	• Sub-component integration: Responsible partners for the component in test • Component integration: FRAUNHOFER with the assistance from component developers
System	• Select test cases according to the requirements • Prepare test beds and test data (if necessary) • Run test beds	• Scripted tests run continuously when components are built on the CI Server • Manual test before installing a new prototype	• FRAUNHOFER with the assistance from component developers

5.2 Integration Plan

Software integration in CPSwarm involves source code and configuration management as well as automatic continuous integration. Existing tools in these domains will be utilized in CPSwarm to improve development efficiency.

5.2.1 Version Control and Issue Tracking

Managing code for a collaborative project requires tools to support parallel and distributed development. In CPSwarm, we utilize Git for source code management. Git is a version control system that tracks changes in computer files and coordinates work on those files among multiple people. It has been widely adopted in the software development world to enable source code versioning and collaboration between multiple developers. As a distributed revision control system, it is aimed at speed, data integrity, and support for distributed, non-linear workflows, which fits the need of CPSwarm. As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

In the CPSwarm project, we will use a Git infrastructure offered as part of the Gitlab software setup and hosted by ISBM. Beside version control, Gitlab also offers standard functionalities such as code review, issue tracking, and built-in continuous integration features. The partners in the consortium have agreed to use the provided Gitlab infrastructure for managing source codes related to the CPSwarm workbench.

We create projects and repositories to manage source codes of components and sub-components respectively. However, this is not enforced and partners are given flexibility for managing the organization and structure of their code. Furthermore, we use common version control practices to manage the source codes:

Each repository comprises three parts:

- Branches, i.e. for experimental feature development;
- Tags, i.e. a marker of version numbers; and
- Master, i.e. for the working source code as the current development version.

Master is the stable development branch, where we store a version of the platform that includes tested features. Branches can be used for each individual feature under development, before they get merged into the master. Tag is used to mark specific important points in history, typically it will be used to mark the version number of a component.

5.2.2 Versioning

Since developers work in a distributed manner in CPSwarm, it is important to establish well-defined interfaces between components. However, as the project evolves, the system architecture is also subject to changes and therefore may also bring changes to component interfaces. In order to manage integration of components with ever-changing interfaces more efficiently, it is important that all developing partners follow a versioning schema during development. We chose the *Semantic Versioning*[1] as our versioning schema.

In summary, the *Semantic Versioning* defines that given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format. The version number should be marked with tags.

In context of CPSwarm, the adoption of *Semantic Versioning* means that all components with the same MAJOR version should use the same set of APIs and therefore can be integrated together. For each component, no incompatible API changes are allowed within one MAJOR version. When a new set of APIs are defined by the consortium, the MAJOR version should be incremented. However, developers of each component are free to change the MINOR and PATCH version, as long as the API of that component is still compatible with other components.

5.2.3 Continuous Integration with Bamboo

Bamboo[2] is a professional tool for continuous integration, deployment, and delivery. While being a commercial and well-maintained product, Atlassian[3] offers free licences to projects that are open-source and public. We plan to utilize an open-source Bamboo license issued for the LinkSmart®[4] ecosystem. It is however simple to request additional licenses dedicated to CPSwarm itself or any future stakeholders interested in providing a CI infrastructure for CPSwarm[5].

[1] http://semver.org/
[2] https://www.atlassian.com/software/bamboo
[3] https://www.atlassian.com/company
[4] https://linksmart.eu/
[5] https://www.atlassian.com/software/views/open-source-license-request

Bamboo assists in continuous integration (CI) by connecting to the version control system (Git), performing tests in a variety of forms, and producing reports and notifications. A successful CI process could then be followed by deployment of resulting artifacts on designated target servers and devices.

In the context of CPSwarm, we configure Bamboo to listen to changes in the repositories hosting components source codes. Each change will trigger the chain of tests at different levels (unit, integration, system) in an isolated environment called a Bamboo Agent. An agent is a containerized environment having a toolset (called Capabilities) suitable for compiling and testing the components. We utilize Docker[6] as the container platform to trigger test plans that are isolated from each other and independent of the underlying CI server operating system. The results of tests are submitted to registered parties in form of an instant message or email.

A successful set of unit tests will trigger sub-component integration tests and subsequently the component integration tests. The system testing which takes more time and consumes more resources will be performed periodically; nightly or every weekend depending on the rate of development iterations.

5.3 Integration and Testing Timeline

The CPSwarm project not only develops a distributed platform but is itself subject to a distributed development process, which makes "integration" a central concern. In alignment with the Description of Action, a timeline has been scheduled ensuring a timely testing and system integration of the CPSwarm components (see Figure 4). This timeline would serve as a reference timeframe for the development of different components. For the first integration phase, four components, namely the CPS library, the swarm library, the Modelling tool, and the simulation environment (including the Optimization Tool and the Optimization Simulator) should be ready for initial integration. In the second integration phase, the above-mentioned components as well as Deployment Tool, Abstraction Library and Monitoring Tool should be integrated. At this time, the overall CPSwarm system should be mostly functional. In the final integration step, all components should be updated according to project evolution and the final CPSwarm system should be built up.

Upon reaching these milestones, integration test for each integration iteration should also be ready so that the components can be tested. After one month, all components should pass the integration test, which marks the success of integration.

[6] https://www.docker.com

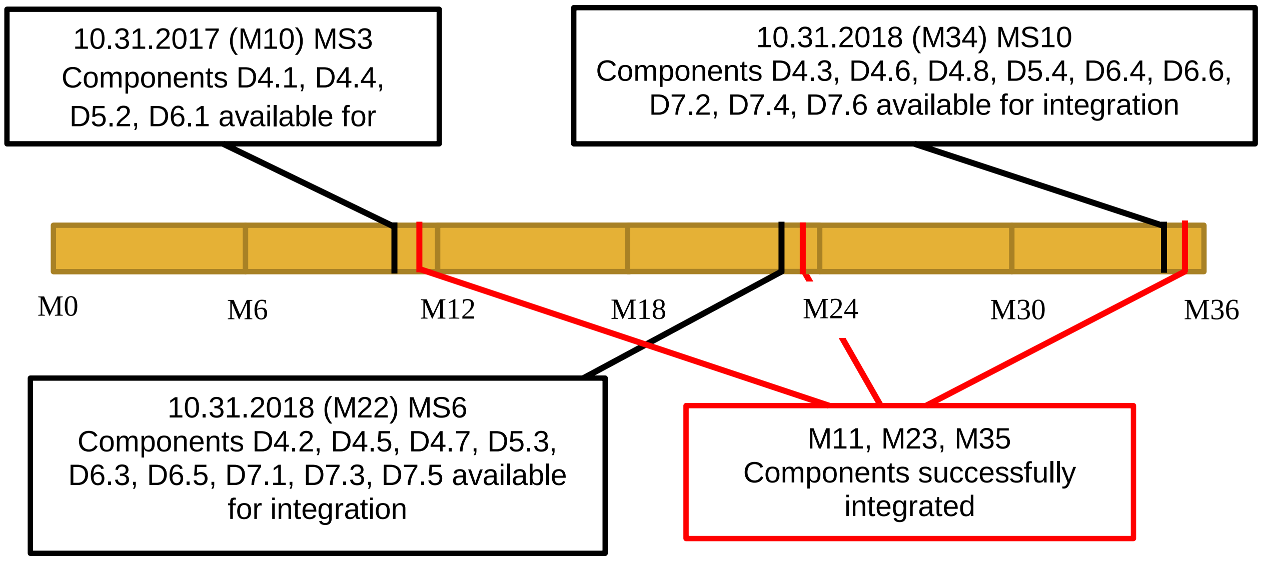
10.31.2017 (M10) MS3
Components D4.1, D4.4, D5.2, D6.1 available for
10.31.2018 (M34) MS10
Components D4.3, D4.6, D4.8, D5.4, D6.4, D6.6, D7.2, D7.4, D7.6 available for integration
M0
M6
M12
M18
M24
M30
M36
10.31.2018 (M22) MS6
Components D4.2, D4.5, D4.7, D5.3, D6.3, D6.5, D7.1, D7.3, D7.5 available for integration
M11, M23, M35
Components successfully integrated

Figure 4 – CPSwarm integration timeline

6 Conclusions

This document introduced fundamental concepts, principles, and standards of software testing and integration. These principles were used as the base to the presented test and integration plan of the CPSwarm software components. This plan, along with the given timeline will serve as the reference for future development activities. All CPSwarm partners should put emphasis to follow the plan in order to ensure high-quality software delivery.

As a next step, this plan will be put into action for the first initial integration phase. Before the end of month 11 (30.11.2017), integration test for multiple initial components (the CPS library, the swarm library, the modelling tool and the simulation environment) should be ready and these components should be put into test. A successful integration is to be expected until the end of month 11. The integration result will be documented in deliverable 3.4 (Initial CPSwarm workbench and associated tools).

As CPSwarm adopts agile development process, the system architecture is subject to future changes. The test and integration plan relies heavily on system architecture and component interfaces and therefore should be revisited, revised, and modified as the project evolves to fulfil the actual development requirements.

Acronyms

Acronym	Explanation
CI	Continuous Integration
CPS	Cyber Physical System

List of figures

List of tables

7 Bibliography

Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques.* International Thomas Press.

Boehm, B. (1989). Software risk management. *ESEC'89*, pp. 1-19.

Bruegge, B., & Dutoit, A. A. (1999). *Object-oriented software engineering; conquering complex and changing systems.* Prentice Hall PTR.

IEEE. (2011). ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.

ISO/IEC. (2011). Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. Technical Report ISO/IEC 25010:2011(E).

Kaner, C. (2006). Exploratory Testing. *Quality Assurance Institute Worldwide Annual Software Testing Conference.* Orlando.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality. volume i. concepts and definitions of software quality.* GENERAL ELECTRIC CO SUNNYVALE CA.

McGregor, J. D., & Sykes, D. A. (2001). *A practical guide to testing object-oriented software.* Addison-Wesley Professional.

Spinellis, D. (2003). *Code reading: the open source perspective.* Addison-Wesley Professional.

Van Ommering, R. (2003). Configuration management in component based product populations. *Lecture notes in computer science*, pp. 16-23.

Weischedel, G. (2013). *Konfigurationsmanagement.* Springer-Verlag.