SOCIETIES
Deliverable D7.1

Initial Integration and Test Plan of the SOCIETIES system

<table>
<thead>
<tr>
<th>Editor:</th>
<th>TRIALOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable nature:</td>
<td>R=Report</td>
</tr>
<tr>
<td>Dissemination level: (Confidentiality)</td>
<td>PU</td>
</tr>
<tr>
<td>Contractual delivery date:</td>
<td>M12 (Sept 2011)</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>M13 (Oct 2011)</td>
</tr>
<tr>
<td>Suggested readers:</td>
<td>WPL, Developers, Integrators of the SOCIETIES Platform</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>36</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Integration, Testing, Development, Versioning,</td>
</tr>
</tbody>
</table>

Abstract

This document gives an initial guideline of the integration and test process of the SOCIETIES Platform. The process is depending on the SOCIETIES architecture and design. The Application Architecture imposes the presence of several deployment nodes or platforms. The Design defines a large number of components to be installed on the different platforms. Therefore the integration and test process shall take into account at least two dimensions: platforms and components. The document describes the integration and test methods and tools for each level of integration: sub-components, components, platforms, and system.

[End of abstract]
Disclaimer

This document contains material, which is the copyright of certain SOCIETIES consortium parties, and may not be reproduced or copied without permission.

_In case of Public (PU):_
All SOCIETIES consortium parties have agreed to full publication of this document.

_In case of Restricted to Programme (PP):_
All SOCIETIES consortium parties have agreed to make this document available on request to other framework programme participants.

_In case of Restricted to Group (RE):_
All SOCIETIES consortium parties have agreed to full publication of this document. However this document is written for being used by <organisation / other project / company etc.> as <a contribution to standardisation / material for consideration in product development etc.>.

_In case of Consortium confidential (CO):_
The information contained in this document is the proprietary confidential information of the SOCIETIES consortium and may not be disclosed except in accordance with the consortium agreement.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the SOCIETIES consortium as a whole, nor a certain party of the SOCIETIES consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Impressum

[Full project title] Self Orchestrating Community Ambient Intelligence Spaces
[Short project title] SOCIETIES
[Number and title of work-package] WP7: Integration and testing (Internal Validation)
[Document title] D7.1 Initial Integration and Test Plan of the CSS system
[Editors: Name, company] Frédéric Mesureur, Bruno Jean-Bart (TRIALOG)
[Work-package leader: Name, company] Bruno Jean-Bart (TRIALOG)
[Estimation of PM spent on the Deliverable] 5

Copyright notice

© 2011 Participants in project SOCIETIES
Executive Summary

This document gives an initial guideline of the integration and test process of the SOCIETIES Platform and consists of four parts.

An introduction defines the role of the Task 7.1 and its links with the other tasks of the WP7. Task 7.1 aims at integrating the components provided by WP4 and WP5. Task 7.2 aims at defining and performing the integration and test for the 3rd party service provided by WP6, while Task 7.3 defines the test bed on which the software will be tested.

Next, the integration process is described including, the role of the actors in the process, the different types of integration and testing phases, and the principles that the project will follow during the integration and test tasks. The process depends heavily on SOCIETIES architecture and design, which come from WP3-6. SOCIETIES Application Architecture, from Task T31, specifies the deployment in terms nodes or platforms. The design deliverables from WP4-6 define the components to be installed on the different platforms. Based on these aspects of the project, the integration and test process shall take into account at least two dimensions: platforms and components.

This deliverable also describes the integration and test methods and tools for each level of integration: sub-components, components, platforms, and system, and provides a first set of test purposes for the main components identified at that time.

The deliverable ends with an overview of the next steps of the integration and test process.
## List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIALOG</td>
<td>Frédéric Mesureur, Bruno Jean-Bart, Olivier Maridat</td>
</tr>
<tr>
<td>HWU</td>
<td>Sarah Gallacher</td>
</tr>
<tr>
<td>ICCS</td>
<td>Ioanna Roussaki</td>
</tr>
<tr>
<td>LAKE</td>
<td>Liam Marshall</td>
</tr>
<tr>
<td>PTIN</td>
<td>Sancho César Rego</td>
</tr>
<tr>
<td>INTEL</td>
<td>Alec Leckey</td>
</tr>
<tr>
<td>IBM</td>
<td>Neta Aizenbud-Reshef</td>
</tr>
<tr>
<td>ITSUD</td>
<td>Haoyi Xiong</td>
</tr>
</tbody>
</table>
1 Table of Contents

Executive Summary .............................................................................................................. 3
List of authors ......................................................................................................................... 4
1 Introduction ......................................................................................................................... 7
2 The Integration Process ....................................................................................................... 8
  2.1 Role of the different WP for the Integration ................................................................. 8
  2.2 Impact of the Architecture on the Integration Phase .................................................... 9
    2.2.1 Deployment Platforms ......................................................................................... 9
    2.2.2 Component integration ...................................................................................... 9
    2.2.3 Focus of integration ......................................................................................... 9
  2.3 Guiding the Development Process ................................................................................. 11
    2.3.1 Software Architect ......................................................................................... 11
    2.3.2 Software Developer ......................................................................................... 11
    2.3.3 Unit Tester ....................................................................................................... 11
    2.3.4 Committer ....................................................................................................... 12
    2.3.5 Integrator ......................................................................................................... 12
    2.3.6 User ................................................................................................................ 12
    2.3.7 Managers ......................................................................................................... 12
  2.4 Principles of Integration for SOCIETIES ................................................................. 13
    2.4.1 Integration Plan ............................................................................................... 13
    2.4.2 Testing Responsibilities .................................................................................... 14
    2.4.3 Unit Testing ..................................................................................................... 14
    2.4.4 Component Testing .......................................................................................... 15
    2.4.5 Platform Testing .............................................................................................. 16
    2.4.6 System Testing ................................................................................................ 17
3 Methods and Tools ....................................................................................................... 18
  3.1 Unit Testing ................................................................................................................ 18
    3.1.1 JUnit on Eclipse .............................................................................................. 18
    3.1.2 Android Testing .............................................................................................. 21
  3.2 Platform and System Tests ......................................................................................... 22
    3.2.1 Integration Test Tools ..................................................................................... 22
    3.2.2 Test Suite Structures ...................................................................................... 24
  3.3 Version management ................................................................................................. 24
  3.4 Bug reporting and Management .................................................................................. 26
4 Initial Testing Suites .................................................................................................. 27
  4.1 Basic Interconnection Tests ....................................................................................... 27
    4.1.1 Installation of the SOCIETIES Platform .......................................................... 27
    4.1.2 CSS Creation .................................................................................................. 27
    4.1.3 CSS Service Discovery ................................................................................... 28
    4.1.4 Device Management ....................................................................................... 29
    4.1.5 Personalisation ............................................................................................... 29
    4.1.6 User Agent ...................................................................................................... 30
    4.1.7 CIS Creation .................................................................................................... 31
    4.1.8 Orchestrate a Community ............................................................................... 31
    4.1.9 SN Connector .................................................................................................. 32
    4.1.10 Context .......................................................................................................... 32
    4.1.11 Privacy ......................................................................................................... 33
    4.1.12 Trust ............................................................................................................. 34
  4.2 Capability Tests ........................................................................................................... 34
  4.3 Behaviour Tests .......................................................................................................... 34
  4.4 Performance Tests ...................................................................................................... 34
  4.5 Scalability Tests ......................................................................................................... 34
  4.6 Robustness Tests ...................................................................................................... 34

© SOCIETIES consortium 2011
1 Introduction

The high level objectives of Task 7.1 are:

- To provide an integration plan of the SOCIETIES platform (encompassing WP4 and WP5).
- To manage the integration and testing process of the SOCIETIES platform.
- To perform the mandatory integration tests in order to provide a robust system before the user trials (WP8).

The SOCIETIES platform includes all core software components of the SOCIETIES system as provided by WP4 and WP5. Scenario-specific services, from WP6, will be integrated and tested in Task 7.2. Tasks 7.1 and 7.2 will use the test bed provided by Task 7.3. Figure 1 below aims at defining the roles of the constituent tasks of WP7 and their relationships.

This deliverable describes the initial integration plan which is defined early in the project in order to organize the software integration. This initial test plan is based on the requirements and architecture respectively defined by WP2 and WP3, and the first design documents provided by WP4 and WP5.

The purpose of having an early integration plan is to anticipate as much as possible the impact of integration in the design and development phase. This facilitates the integration process since the cost of fixing an error is lowest in the first phase of system development.
2 The Integration Process

2.1 Role of the different WP for the Integration

This section aims at defining the general overview of the integration process within SOCIETIES.

- The tasks of integration are shared between WP7 and WP4, WP5, WP6, WP8.
- A strong coordination is needed between these WPs in terms of methods, tools, outcomes.

The following diagram depicts the relationship between WPs in terms of integration.

![Diagram showing the relationship between Work Packages for Integration]

**Figure 2: Links between Work Packages for Integration**
2.2 Impact of the Architecture on the Integration Phase

2.2.1 Deployment Platforms

The WP3 Application Architecture [1] defines several types of deployment platforms. This architecture is the main driver for the integration plan as it has a strong impact on the integration: the SOCIETIES software is distributed among several hardware entities as depicted in the picture below. Therefore the integration will be performed at two levels and following several dimensions.

![Deployment Platforms hierarchy diagram](image)

Figure 3: SOCIETIES Deployment Platforms

2.2.2 Component integration

The term component will be used to represent how integrators want to relate to software. This definition is provided by Ralf Johnson and quoted in the book [2]. “Customers want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers”.

Components will be provided by WP4,5 software developers [3, 4]. Each component exposes its external interfaces as “provided interfaces” when the interfaces are supplied by this component or as “required interfaces” when the component will need to access to the external interfaces of another component. A component consists of sub-components, each sub-component to be developed and unit tested by an identified software developer. Interfaces between sub-components are called “internal interfaces”.

External interfaces can be public in the sense they are available to 3rd party services. External interfaces are private when they shall be used only by SOCIETIES components.

2.2.3 Focus of integration

As already indicated, the integration process consists of two dimensions, the platform and the software components provided by the WP4, WP5 and WP6.

Each work package aims at providing different types of software (WP4: platform, WP5: user experience, WP6: 3rd party applications) and the component dimension will be structured by component dependencies instead of WP4,5 structures. This component dimension will be studied in collaboration between WP4, WP5, and WP7. As concerns the 3rd party applications, they will be integrated separately in Task 7.2.

Therefore, this deliverable focuses on integration of WP4 and WP5 components. For the initial phase, WP4 and WP5 will work mainly on three platforms: Participant Light Platform, Participant Rich Platform,
Participant Server Platforms, respectively including WP4 and WP5 software of the Light Client Node, Rich Client Node, Cloud Node defined by WP3 Architecture as follows:

**Definitions**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Light Client platform</strong></td>
<td>The light client platform includes the services and components necessary to allow a user to interact with the SOCIETIES system. It contains the minimum core services required, and corresponding management GUIs. It is designed to operate on a smart phone or lower specification tablet PC. Thus it is envisioned that these components are customised to the mobile OS on which they run, have limited battery life, are portable enough to be carried with the end-user.</td>
</tr>
<tr>
<td><strong>Rich Client platform</strong></td>
<td>The rich client platform includes the services and components necessary to allow a user to interact with the SOCIETIES system in a feature rich way. It contains the core services required, and management GUIs needed to operate on a desktop computer, laptop, interactive display, or tablet PC. It is envisioned that this platform has the ability to provide limited third party services tailored to the environment they operate in.</td>
</tr>
<tr>
<td><strong>Cloud platform</strong></td>
<td>This platform includes the core services that operate on behalf of a participant running on a cloud or server node. It contains also all the services necessary to support the creation, management and maintenance of a CIS. In other words these services act on behalf of a CIS to provide benefit for all members of a CIS. These services are placed into this platform as they require access to a large volume of participant data, or need to run for an extended period, or be always available.</td>
</tr>
</tbody>
</table>

The following diagram shows the different blocks of components identified by WPx yP where x is 4 or 5 and y is Light (L), Rich (R) or Cloud (C). There are 6 blocks of components to integrate at the platform level, followed by testing at a system integration level. In the picture the overlap between WP4 and WP5 is due to the fact that relations exist between the software from these workpackages.

Another dimension to take into account is for the Cloud platform where several deployment packages could exist in the future depending on the features required. The diagram represents that fact with several WPx CP represented for Cloud platform.

![Figure 4: The Three dimensions of Integration](image-url)

**Figure 4: The Three dimensions of Integration**
2.3 Guiding the Development Process

SOCIETIES is an Open Source project and WP7 will provide guidelines to support this. Each partner of the project will have a specific role as contributor and all contributions will have to be recognized and validated before any integration process.

From the integration and testing point of view, the contributors can be classified as:

- software architect,
- software developer,
- unit tester,
- committer,
- integrator,
- user,
- manager.

According to the description of work of the SOCIETIES project, software developers, unit testers and committers are WP4, WP5 and WP6 partners, the integrators are WP7 partners and the users are WP8 partners.

2.3.1 Software Architect

The software architect is in charge of the global software architecture and he is responsible for defining the software structure, i.e. the software breakdown into software components and sub-components as well as the execution environment (real-time architecture with a real-time operating system, for example). Command and data flows between the components are also identified.

To do that, the architect has to allocate the functional requirements to software components and ensure that this allocation is consistent and fulfils the functional requirements. Non-functional requirements are also taken into account by the architect for the construction of the software.

The software architect will provide a System Architecture (WP3) document that will be used to allocate the work to the software developers (who will design the software components).

This document will also be used by the integrator for the understanding and the construction of the software.

2.3.2 Software Developer

The Software Developer (SD) is responsible for the specification, the design and the coding of software sub-components. This work is performed within the WP4.

The SD uses the Software Design document to identify the functionalities allocated to the sub-component. He also uses this document to identify the interaction of his sub-component with other sub-components.

To design a sub-component, the SD has to:

- Write a Software Design document (detailed design)
- Write the Source Code
- Test the component
- Write the software sub-component release note

2.3.3 Unit Tester

The Unit Tester (UT) is responsible for the Unit testing of the software sub-component. To perform that unit testing, the UT writes a unit test plan based on the Software Design Document of the sub-component. He then writes the test scripts, executes the tests and reports the results to the SDs. Test scripts and Test Reports are committed to the versioning management system. The Unit Tester and the Software Developer can be the same person.
2.3.4 Committer

The committer is the person in charge of integrating the sub-components and delivering the resulting component to the platform integrator. Committers are responsible for the review of software sub-components. A successful review consists in checking conformance to coding and documentation standards as well as test coverage. The committer will have to perform component testing before software can be committed. The committer is appointed by WP4 or WP5.

2.3.5 Integrator

The integrator will test and validate the interactions between components on a dedicated platform or interaction between platforms by specifying, implementing and executing integration tests. The integrator will analyse how to integrate the components by specifying the integration test plan. He will produce test purposes, implement the test cases then execute the integration tests on the project testbeds. The delivery of the integrator includes:

- Integration test plans,
- Integration test reports,
- Bug reports
- Complete integrated software

An integrator is a person, inside the SOCIETIES project, which is appointed by WP7.

2.3.6 User

The user is anyone who will use the integrated and tested SOCIETIES platform software in particular for field operational tests. The user will report faults and bugs by using the bug management tool set up for integration and validation phase. If the user (e.g. in Enterprise group) cannot report faults itself, the system shall be instrumented to detect them.

2.3.7 Managers

The managers form the Change Control Board (CCB) structure to validate the bugs tracked during the integration process. The CCB will include when necessary other types of contributors (developers, integrators) with the objectives to discuss the open issues concerning bugs and change requests SOCIETIES. This CCB is responsible for setting its overall integration direction.
2.4 Principles of Integration for SOCIETIES

This section describes the approach that will be taken for the software integration of the SOCIETIES platform. The software integration is a process in which all the contributors listed in section 2.3 are involved.

2.4.1 Integration Plan

The Integration Plan aims at providing the global coordination of the integration. The integration plan is illustrated in the diagram 5 below. Key elements of the integration process are:

- Sub-component refers to the software unit developed by a software developer (e.g. CSS Discovery).
- Component refers to a set of sub-components integrated together to provide a deployable piece of software (e.g. CSS Management as a jar file).
- Platform refers to a set of components integrated within a specific node (e.g. Rich client platform)
- System refers to a set of platforms collaborating to provide the SOCIETIES features.

Integration steps start with the software developer and finish with the system integrator. The process is driven by a set of test plans defined at the different phases. Each test plan aims at providing the proof that the next step can be performed. The first two steps are appointed to WP4 and WP5: apart the development itself, WP4 and WP5 are concerned with the unit testing, the component integration, and component testing.

Figure 5: The Four Different Phases of Integration and Tests

Once a component has been tested, he can be committed to the Integration branch (see section 3.3). The WP7 role is to coordinate the whole process but WP7 partners will be mainly involved in the platform and system integration phases and in consequence will work only on software committed in the Integration branch.

To perform platform integration, the WP7 expects the following contributions from WP4 and WP5:

- The component external interfaces: these interfaces will be grouped together in a specific component. This specific component will be tested independently to check APIs integration first. This component will be further used as APIs reference for all other components.
The component themselves once they have tested by the committer.

- The component test plan and test reports

### 2.4.2 Testing Responsibilities

The table below provides the role of integration and tests for the actors:

<table>
<thead>
<tr>
<th>Test Phases</th>
<th>Contributor</th>
<th>Work Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Testing</td>
<td>Software Developer, Unit Tester</td>
<td>WP4, WP5</td>
</tr>
<tr>
<td>Component Testing</td>
<td>Software Developer, Committer</td>
<td>WP4, WP5</td>
</tr>
<tr>
<td>Integration Testing</td>
<td>Integrator</td>
<td>WP7</td>
</tr>
<tr>
<td>System Testing</td>
<td>Integrator</td>
<td>WP7</td>
</tr>
</tbody>
</table>

**Table 1: Role of the Contributors on the Integration**

Due to the importance of testing in the integration process, the following sections define the principles of test processes to be applied at each integration phases.

### 2.4.3 Unit Testing

Unit testing is the test method to be used by the developers of individual software modules. Unit testing is required before the integration of the sub-components within a component. This testing phase is mandatory for all sub-components with the following recommendations.

Testing practices at that level are:

- Write unit tests that will return information.
- Do not create test cases necessarily for all methods.
- Keep unit tests present with your development and add new tests after problems reported by the module integrator.

JUnit is the original Java unit testing framework that has given rise to many imitators and programming/OS implementations generally known as the xUnit family. Kent Beck created sUnit for Smalltalk and he then ported it to Java with Erich Gamma. Unit testing forms part of the concept of Test Driven Development (TDD) where classes are individually tested with automated tests to encourage frequent and repeatable testing, to discover problems as early as possible, and ensure that class methods and code branches are tested to a required level.

The main concept behind JUnit is the Test Suite. A typical JUnit class test consists of one or more test methods that exercise some method(s) in the target class. A Test Suite is a collection of tests. It is also possible to create pre and post methods to allow initialisation and tear down conditions for the running of the tests. The individual tests involve using JUnit assertions to check and validate that the test code is reaching certain expected states and outputting certain results for defined inputs.

It is not an infallible technique and it does require that the developer uses their knowledge of the class(es) in question to exercise the code and functionality as well as possible. Other tools such as static code analysers, style checkers and code coverage aid the basic technique.

Some questions that are regularly raised are:

**Why do I have to write nearly as much code to test the class as the class contains?** For some classes the JUnit test code may exceed the class code though in general it is usually less and depends on the level to which the class can be exercised and to what degree the developer/project requires test thoroughness. The JUnit code is just Java code like the original classes to allow a concise set of tests to be constructed by the same developer(s) as created the original code - it is a developer's natural test language. The motivations behind developing the tests are many and the investment in developing the tests and their continual maintenance may not be apparent at first:

- class code is tested preferably by the original developer - they understand the target class code the best
• the tests can be developed incrementally and when using IDEs such as Eclipse with JUnit integration the process is simple and the feedback quick - "write code, create/amend test, run test, view result, repeat"
• running the tests prior to any code changes and/or repository commits provide a level of assurance and more importantly ensure that regression testing is carried out on all relevant classes to ensure that code changes have not impinged on other classes
• refactoring of code due to code optimisation changes can be carried out with confidence as the functionality will be tested consistently before and after the refactoring
• the tests act as tutorials to other developers in how to use the target classes
• code changes due bugs/issues and their preceding cause can be captured as tests to ensure they are tested in future releases
• bad or inefficient class APIs will be detected and confronted by the target class developer and not by other developers

Why do I need to test at all? In the Agile/Test Driven Development world, code is worthless or not considered to be code unless it is tested. In the older waterfall methodologies, programmers developed and testers tested. This resulted in long and unpredictable development times, developers evading responsibility for their own code and chaotic results when modifying codebases. Developers come and go and unless tests can be developed that encapsulate code knowledge and state changes, the learning curve for new developers and testers becomes steep and code entropy accelerates. The ability to automate unit testing in a predictable and reproducible way gives developers and other project personnel a valuable means of engendering confidence in the code base. However the principle of GIGO (garbage in, garbage out) still applies and unit testing is only useful if all developers accept the principle and become enthusiastic in its practice.

Will the tests become as much a maintenance task as the codebase itself? The hardest part is the creation of the original tests. Amending tests will involve less work although it should always be accepted that there is a limit to continual test modification without finally evaluating the efficacy of the test and possibly re-designing it. Depending on the amount of class linkage in a module, the maintenance can be high or low when code changes break certain tests and ripple through to other tests. Minimising class linkage and ensuring simple focused classes is a general software design aim and not the fault of unit testing. In fact, unit testing will help make these deficiencies more visible.

For more information please consult:
• JUnit website (http://www.junit.org/) [5]
• JUnit anti-patterns [6]
• JUnit Book [7]

2.4.4 Component Testing
Testing a component is a further extension of unit tests. The person in charge of this test can be different from the component developer.

Component testing is the test method to be used by all the committers (i.e. component integrators). Component testing is required before the platform integration. This testing phase is mandatory for all components with the following recommendations.

Mocking Framework
Mocking frameworks allow the developer to create stubs/mock classes dynamically from target classes/interfaces and ensure that unit testing is only concerned with the main class in question and not creating sub-module tests that more functionality tests rather than unit tests. Sometimes class dependencies do not exist due to development priorities, are complex classes whose initialisation involves their dependencies being satisfied, are part of a container whose initialisation is not possible and/or realistic for the purposes of unit testing and other reasons may make the inclusion of these ancillary classes in the unit testing process difficult or impossible.

This is where mocking frameworks come to the rescue and allow the developer to fake dependency classes with the least effort and more importantly prevent the creation of classes that mimic real classes with all of the maintenance and work issues that these extra classes entail. The mocking framework dynamically creates
the relevant class dependencies and the developer injects the least amount of class behaviour to enable the unit test to complete. The unit test is where this class behaviour is inserted and not some other entity.

Not all class dependencies need to be mocked - closely linked classes should be used to ensure that actual behaviour is tested. The decision on when to use mock classes/objects is one that the developer needs to learn and strike a balance between being practical and being ideological. Good uses of mock classes/objects are:

- Complex class dependencies that detract from the unit testing and makes the test more a functional test
- Class dependencies that deal with unpredictable behaviour such as communication, sensors and operating system resources
- Class dependencies that are part of another class library that is expensive to use
- Class dependencies that are part of a container, e.g. OSGi and Android

Some well known frameworks are:

- EasyMock : [8]
- Mockita : [9]

The choice of one of these or another framework will be made in the next weeks.

2.4.5 Platform Testing

The principle of the platform testing is to use a remote Tester Tool to perform a black box testing of the System Under Test (SUT). The SUT represents the software to be tested which runs in its target platform. In SOCIETIES there will be several SUTs, at least one per deployment platform (Light node, Rich, Cloud nodes).

The approach is depicted in the figure 6. We will have a Test Tool and test suite per WPx yP families.

![Figure 6: The Remote Test Principle](image)

The Test Tool runs a set of test suites. The Test Suites will be written typically using a Test Language, e.g. for instance such as TTCN-3 [10]). Apart from the User Interface, a Test Tool Architecture (figure 7) consists mainly of three components that are important to describe here:

- The Test Engine is the software that runs the sequences of actions defined in the test suites scripts. The Test Engine is a generic component, independent from any system under test.
- The Upper Tester is a component to be developed specifically to access the upper part of the SUT using the communication protocols imposed by the latter. The upper part is typically the User Interfaces, or highest layer of a protocol stack for instance.
- The Lower Tester is like the Upper Tester. The only difference is that Lower Tester provides access to low-level part of the SUT such as data sensors, communication from another node, etc...
2.4.6 System Testing

The same remote testing approach as the one described in the section above will be used for system testing. The difference with the platform testing is that the SUT will include several nodes in parallel, i.e. Light and Cloud Nodes. In consequence the Remote Test Tool shall manage several Upper and Lower Testers. During System testing, the focus will be mainly on performance tests rather than functional tests.
3 Methods and Tools

This section describes the two types of testing methods used during the development and integration phase:

- Sub-Component and Component testing that will use the Unit and Mocking Testing methods;
- Integration of Platform and System testing.

At the end of the section, the versioning and bug tracking process will be presented.

3.1 Unit Testing

3.1.1 JUnit on Eclipse

Creating your own Test Class

Eclipse offers a tool to generate test classes for you.

1. Right click on the folder where you will place your test. The window of the figure 8 below will appear.
2. Select New/Java Test Case (if this option is not available, click New/Other, and then select 'Java Test Case' from there.

![Figure 8: Eclipse Menu to create a Unit Test Case](image)

3. You will be presented with the following screen. Fill in the required details:
a. You need JUnit 4 for Unit testing. Tests that run using Knopflerfish JUnit bundle tool will have to be JUnit 3, as that is the version used by Knopflerfish (until such time as a work around is found)

b. Most test classes will have a setup() and teardown() method to ensure that each test is run with the same state (and not within a cumulative state)

c. Select the class you are testing from ‘Browse

4. Select the methods from the class you wish to test
5. You will get a generated class as follows:

```java
package org.personalsmartspace.examples.printservice.impl;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class MyTestCase {
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testPrint() {
        fail("Not yet implemented");
    }
}
```

a. Add code to the `setUp()` that should be executed at the start of each test to set it up (e.g. instantiate a variable)

b. Add code to the `tearDown()` that should be executed at the end of each test to bring everything back to an initial state, so further tests can be run in isolation.

c. Add the test code to `testPrint()` to call the `print()` method.

Note: In more complex settings you may need to use stubs for components to which you have external dependencies. Alternatively, use a tool such as JMock (http://www.jmock.org/), Mockito (http://mockito.org/) etc. Component stubs are being written as part of the integration testing. In general it is a good idea that you provide stubs to your own components, so developers are not waiting on the real thing before they can do some level of integration.
3.1.2 Android Testing

3.1.2.1 Android Virtual Machine

Although Android uses Java as its native programming language (C and/or C++ can also be used for more specialised applications using the NDK: http://developer.android.com/sdk/ndk/overview.html), it should be emphasised that the normal JVM is not used either on the Android Emulator or actual devices. Instead, the Dalvik Virtual Machine (DVM) is used which uses the DEX format for classes and different byte instructions than those used in the JVM. When an Android application is created, the application's Java classes (.class) are converted to .dex files which use the DVM byte code and cannot be run in a JVM. The main reasons for the creation of the DVM were efficiency and optimisation for constrained devices.

It should also be noted that the Android Java libraries do not correspond to the standard Java SE/ME libraries but are based on a subset of the Apache Harmony (http://harmony.apache.org/) libraries. In practice, this difference is negligible but notable exceptions are the AWT and Swing libraries.

3.1.2.2 Conventional Android Unit Testing

In order to test standard native Android applications, an Android Test application can be created in Eclipse (cf. http://developer.android.com/guide/topics/testing/testing_android.html) and linked to the relevant Android application. Activities, Content Provider and Services can be tested using conventional JUnit 3 test classes/suites. Some mock Android classes are also available to aid testing. One important point to note is that the test runner runs on either the relevant emulator AVD and/or connected Android device and not in Eclipse. This can lead to longer initiation and test run times than normally experienced in conventional Java development.

3.1.2.3 HTML5/Javascript/CSS and PhoneGap Testing

A much more challenging case for testing arises when using Webview containers embedded in standard or PhoneGap enabled applications.

3.1.2.4 Plain HTML5 development

While the Java portions of the application be tested using the conventional testing approach, the HTML5/Javascript/CSS code will require other approaches adopted from Web technology development. Tools such as JSLint (http://www.jslint.com/), the Firefox and Chrome browser plug-in Firebug (http://getfirebug.com/) and Chrome Developer tools can be used to validate, test and debug.

The major problem encountered will be debugging the code on an AVD or device. One partial solution is Weinre (http://phonegap.github.com/weinre/) but it does not allow remote debugging. Another is the use of logging which can be filtered and reviewed in the Android logcat utility (Eclipse DDMS or command line).

3.1.2.5 PhoneGap development

Since the majority of code development will be in HTML5, Javascript and CSS the comments in the previous section will apply. One area not covered will be the development of PhoneGap plugins which are Java classes. These classes are challenging to test as they do not seem to be able to be tested using the conventional Android testing framework and so must be tested using conventional JUnit tests. This approach quickly runs into problems as the plug-in classes cannot be tested without an Android Jar file to either allow direct usage of Android classes or to use in a mocking framework such as EasyMock (http://easymock.org/) or Mockito (http://mockito.org/).

It might be assumed that the Android SDK contains the relevant Android Jar files and it does. However, the classes contained in the Jar file have all of their methods declared as final and all throw exceptions when an attempt is made to use the class methods. As a result, direct usage of the Android libraries is prevented and conventional mocking frameworks cannot be used. One framework that may work is PowerMock (http://code.google.com/p/powermock/). The use of a Java decompiler tool such as JAD (http://www.varaneckas.com/jad) or JD (http://java.decompiler.free.fr/) will confirm the above observations.

The following approach can be used to solve the above problem:

Extract a "real" Android Jar file from the Android Emulator or a rooted device using the ADB utility

Convert the DEX format to Java classes using a tool such as dex2jar (http://code.google.com/p/dex2jar/)

© SOCIETIES consortium 2011
Use the resulting Android JAR file to link against and/or use with a mocking framework. The second option is recommended as the direct usage of Android classes can result in complicated initialisations.

One side-effect of the above approach is that the public and hidden Android APIs will be exposed for inspection.

### 3.2 Platform and System Tests

The testing method for platform and system tests will be based on the ISO/CEI 9646 standard [2].

The goal of the ISO/CEI 9646 [11] is to define, at an international level, a common testing methodology, relevant methods and testing procedures, a general frame for the specification of the test suites and the procedures to follow during the tests.

The conformity trials include the capacity trials and behaviours trials of a system. The observations must be compared to the conformity conditions specified in the specifications of the system and to the capacity to the system to be tested. Although the ISO/CEI 9646 focuses on conformity trials, this testing method provides the ability to expand the test coverage to the evaluation of the functionalities, performances, robustness or interoperability of a system. The conformity evaluation of a system associated to the evaluation of its performances, robustness or its interoperability will guaranty a perfect integration of the system.

The tests orientation should be orchestrated by isolating the functional chains involving several architectural components to verify the integration and interoperability into the system to validate.

This identification of the functional chains will be done based on a bottom-up approach. Architectural elemental components with one or several simple functional interactions will be identified, integrated and tested to guaranty a perfect integration. Complex architectural components, based on the association of functional elemental architectural components already integrated will be, in turn, integrated and tested. By the progressive inheritance of integrations and tests, we will be comfortable to guaranty, to the end, the total integration of the system.

#### 3.2.1 Integration Test Tools

The test tool for remote integration testing will be based on the Trialog Testing Tool and depicted in the figure 11 below. This tool has a generic HMI and a generic test engine. This test engine is able to run test script written in TTCN3.

The principle of the tool is that the specific part of the testing software is distributed into proxy libraries. The libraries implement the upper and lower testers of the general ISO 9646 architecture to access the System Under Test. In our case, these libraries will be typically communication components using XMPP, REST, HTTP protocols.

A simple script example is provided below. This script aims at testing whether the SUT will return a CODE_OK when a plc_DiscoveryStart call is applied. The verdict test is simple here. The plc_DiscoveryStart () function is provided by the plc library and depend upon the SUT (in this case the SUT is a device communicating over Powerline (plc)).

```plaintext
/**
 * Call Plc_DiscoveryStart
 * @copyright Trialog 2011
 **/
module Plc_DiscoveryStart {
    /******************* CONSTANTS ******************/
    import from apik_constants { const all }
    import from manual_action_constants { const all }
    /***********************FUNCTIONS ***************/
    Control {
        apik.start;
        var integer returnType;
        returnType := plc_DiscoveryStart( 1 );
        if( returnType !:= RETURN_CODE__OK ) {
```
logString("Plc_DiscoveryStart failed");
    return 0;
}
logString("Plc_DiscoveryStart successful");
apik.stop;}}

Figure 11: Trialog Test Tool HMI.
3.2.2 Test Suite Structures

All integration test suites will have a structure as follows. Tests Cases are organised per category of tests. For each category, the structure will be organised per families of functionalities of the SUT, following the WP4 and WP5 modules features and capabilities.

- Basic interconnection tests: the goal is to determine whether interconnection is possible at an initial stage. These tests will establish that the main features of the SUT are implemented.
- Capability tests: the goal is to validate that all the observable external static capabilities of the SUT are valid with respect to the static conformance requirements expressed in the definition of the SUT.
- Behaviour tests: the goal is to check the dynamic conformance of the SUT with respect to the dynamic conformance requirements specified in the specification of the SUT.
- Performance tests: the goal is to check the capacity of the SUT when it is submitted to a large number of events.
- Scalability tests: the goal is to check the capacity of the SUT when it is submitted to a large number of users.
- Robustness tests: the goal is to check the capacity of the SUT when it is submitted to stress tests.
- Interoperability tests: the goal is to check the conformance of SUT towards the interoperability architecture.

3.2.2.1 Test template

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Test Name</th>
<th>Test Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>The test ID shall provide a way to identify where the test case is placed in the test suite structure. e.g. Func_Family_Number</td>
<td>A generic expression to identify the function to be tested. e.g. Create a new User.</td>
<td>This field shall provide a summary of the objectives of the test cases.</td>
</tr>
<tr>
<td>Preamble</td>
<td>This part is used to set the SUT into the mandatory state before run the test body. E.g. Populate the SUT with the right data.</td>
<td></td>
</tr>
<tr>
<td>Test Body</td>
<td>This part contains the script of the test itself. The sequence of actions to be performed to the SUT, the response expected from the SUT, the comparison test between the expected responses and the actual responses providing the verdict: PASS, FAIL, INCONCLUSIVE.</td>
<td></td>
</tr>
<tr>
<td>Postamble</td>
<td>This part of the test case is necessary to set the SUT into an idle state so that the next test case can run.</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Version management

Given the size of the project and the number of developers a commercial source control process is proposed. In this process, there are branches as follows:

- **trunk/development** - containing the code that new features are developed on and is continuously integrated. All development is on trunk until the Feature Set Complete Date (FSCD) is reached and then a branch is created and then the development for the next release can continue on trunk.
- **branch/system test** – this is the code that is tagged at the end of feature development and deployed to system test for final system testing and release to trial sites. This is where the code is branched from the trunk. After final system test and bug fixing the code is merged back down to the trunk.
- **production/trial sites** – this is the code that is deployed to the trial sites
Figure 12: Production Release Version Management

**Versions**

- Github Repository generates a version number on every commit.
- Build number generated for every successful build.
- Release Version number is generated when features are ready for final system test and then deployment and this build is tagged with this version number (See diagram above).

**Tagging/Versioning**

- When the features are developed and ready for final System test the build is tagged with the version 0.1 and deployed to the integration testbed.
- At the end of the System testing and with the software ready for deployment to the trial sites, this build is tagged as version 1.0 and deployed to production/trial sites and at the same time the bug fixes that were implemented during this testing phase on the branch/integration code are merged down into the trunk/development branch.
Notes

By following an agile development methodology and using Continuous Integration the integration diagram in section 2.4.1 will be part of the Continuous Integration build process which should incorporate all the testing phases (Unit, Module and Platform) to output a build that is ready to be deployed to the physical testbed.

The Operations Team shall consist of the Trial Site Managers and the WP7 Release Manager and will be responsible for the deployment of the release to the trials sites.

3.4 Bug reporting and Management

The Redmine tool [12] will be used for bug tracking:

The guidelines of the tool are provided by the test bed task T7.3. A concise guide is available in D7.2. To manage conflicts, the WP7 will create a Change Control Board (CCB) structure to validate the bugs tracked on Redmine. The CCB will include several types of contributors (developers, integrators, managers) with the objectives to discuss only the open issues concerning bugs and change requests.
4 Initial Testing Suites

This section provides an outline for the SOCIETIES platform test suites. A placeholder structure for testing suites is outlined, however in this initial integration and test plan, only some test purposes are provided (i.e. the basic interconnection tests). The complete set of test purposes and test cases will be provided in the next months and before the starting of integration and testing phase at Month 18.

The integration tests shall not repeat the components tests: they should be oriented to features visible by the integrators rather than testing each individual components and functions. For instance, there is no specific tests family of the platform infrastructure such as communication. The communication features will be integrated and tested via the other test families.

The test suites follow the structure of the SOCIETIES platform design (D4.1) and the WP5 currently identified components.

4.1 Basic Interconnection Tests

The goal of the Basic Interconnection Tests is to determine whether interconnection is possible at an initial stage. These tests will establish that the main features of the SUT are implemented.

4.1.1 Installation of the SOCIETIES Platform

This family of tests refers to the test cases where the user is performing the first actions related to the installation of its platform.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud client</td>
<td>Test if the SOCIETIES software installs successfully on a cloud node.</td>
</tr>
<tr>
<td>Install societies</td>
<td>This test case aims at checking whether the installation package is</td>
</tr>
<tr>
<td>software</td>
<td>installed on a Linux Server.</td>
</tr>
<tr>
<td>Light client</td>
<td>Test if the SOCIETIES software installs successfully on a light client, i.e.</td>
</tr>
<tr>
<td></td>
<td>an Android device.</td>
</tr>
<tr>
<td>Rich client</td>
<td>Test if the SOCIETIES software installs successfully on a rich client, i.e.</td>
</tr>
<tr>
<td></td>
<td>Virgo platform on a Linux or Windows PC.</td>
</tr>
<tr>
<td>Usability after</td>
<td>After successful installation, test if user can enter his personal data</td>
</tr>
<tr>
<td>installation</td>
<td>and use the SOCIETIES platform. Applies to every supported platform.</td>
</tr>
</tbody>
</table>

4.1.2 CSS Creation

This family of tests refers to the test cases where the user is performing the actions related to creation of its CSS (creation, profile, privacy preferences, etc).

This family should include test cases for the modules CSS Management, CSS Registry.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a node</td>
<td>The purpose is to check the actions performed by the system when the user</td>
</tr>
<tr>
<td>Configure CSS</td>
<td>The purpose of this test is to check if a CSS can be configured.</td>
</tr>
<tr>
<td>Configure CSS Node</td>
<td>The purpose of this test is the check if the CSS Node can be configured</td>
</tr>
<tr>
<td>Test Case</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Creating CSS</td>
<td>Test all the steps of the creation of the CSS.</td>
</tr>
<tr>
<td>List CSS Node</td>
<td>The purpose of this test is to check if result of this function is the list of all the CSS Node added to the CSS.</td>
</tr>
<tr>
<td>Remove CSS</td>
<td>The purpose of this test is to check if the user can delete CSS.</td>
</tr>
<tr>
<td>Remove CSS Node</td>
<td>The purpose of this test is to check if the user can remove CSS Node from his CSS.</td>
</tr>
</tbody>
</table>

### 4.1.3 CSS Service Discovery

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Un)Register services to the Service registry</td>
<td>The purpose of this test case is to test if the SOCIETIES platform is able to support the (un)registration of both 3P and platform services in a service registry that may exist on CSS nodes.</td>
</tr>
<tr>
<td>Service registry queries</td>
<td>The purpose of this test is to check if the SOCIETIES platform is able to retrieve service (3p and platform) related information maintained in a service registry.</td>
</tr>
<tr>
<td>Synchronising service registries on different CSS nodes</td>
<td>The purpose of this test is to check if the SOCIETIES platform is able to synchronise information related with the service registration from various nodes that consist the CSS.</td>
</tr>
<tr>
<td>Intra CSS service discovery</td>
<td>The purpose of this test is to check if a node running the SOCIETIES platform is able to discover services that are provided by another CSS node.</td>
</tr>
<tr>
<td>Inter CSS services discovery</td>
<td>The purpose of this test is to check if a node running the SOCIETIES platform is able to discover services that are provided by a remote CSS node that doesn’t belong on the same community.</td>
</tr>
<tr>
<td>Intra CIS service discovery</td>
<td>The purpose of this test is to check if it possible a CIS to discover services provided by another CIS.</td>
</tr>
<tr>
<td>Inter CIS service discovery</td>
<td>The purpose of this test is to check if the SOCIETIES platform is able to discover services based on various criteria, modelled as context data (e.g. location, service type etc.)</td>
</tr>
<tr>
<td>Context based Service Discovery</td>
<td>The purpose of this test case is to test if the SOCIETIES platform can enable a service to control its visibility at CSS and CIS level, i.e. to control whether it is visible outside of the CSS or outside/within selected CISs.</td>
</tr>
</tbody>
</table>
4.1.4 Device Management

This family of tests refers to the test cases where the user is performing the actions related to device management of its CSS (add, remove, selecting devices, etc).

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discover a Device</td>
<td>The purpose is to check the actions performed by the system when a new Device is discovered.</td>
</tr>
<tr>
<td>Add a Device to the device registry</td>
<td>The purpose of this test is to ensure that a newly discovered device can be added to the device registry</td>
</tr>
<tr>
<td>Remove a Device from the device registry</td>
<td>The purpose of this test is to ensure that a device can be removed from the device registry in the circumstances of the device no longer being available</td>
</tr>
<tr>
<td>Synchronise the Device registries</td>
<td>The purpose of this test is to ensure that the device registries on different devices can be synchronised in order to maintain consistency across the CSS</td>
</tr>
<tr>
<td>Synchronise data between 2 Devices</td>
<td>The purpose of this test is to ensure that the data on different devices can be synchronised in order to maintain consistency across the CSS</td>
</tr>
<tr>
<td>Configure Device</td>
<td>The purpose of this test is to ensure that the device can be configured correctly</td>
</tr>
<tr>
<td>Retrieve Device attributes information</td>
<td>The purpose of this test is to ensure that the device attributes associated with a particular device can be retrieved from that device and displayed to the user and or used by the system.</td>
</tr>
<tr>
<td>Set Device as “Context Source”</td>
<td>The purpose of this test is to ensure that a device can be tagged as a context source for use with the context source manager module</td>
</tr>
<tr>
<td>Send context data to “Context Source Manager”</td>
<td>The purpose of this test is to ensure that a device can be used correctly as a context source by the context source manager module and provide the relevant data</td>
</tr>
<tr>
<td>Change status of Device during runtime</td>
<td>The purpose of this test is to ensure that a device can change it’s status during runtime and this status change will be handled correctly by the system</td>
</tr>
</tbody>
</table>

4.1.5 Personalisation

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preference evaluation</td>
<td>The purpose of this test is to check if the Preference Manager can successfully evaluate preferences and suggest a Preference Outcome to be implemented</td>
</tr>
<tr>
<td>Dianne prediction</td>
<td>The purpose of this test is to check if the DIANNE component can successfully deliver a Dianne Outcome to be implemented</td>
</tr>
<tr>
<td>CAUI prediction</td>
<td>The purpose of this test is to check if the CAUI prediction component can successfully predict a CAUI user intent action to be implemented</td>
</tr>
<tr>
<td>CRIST prediction</td>
<td>The purpose of this test is to check if the CRIST prediction component can successfully predict a CRIST user intent action to be implemented</td>
</tr>
</tbody>
</table>
### Proactive personalisation
The purpose of this test is to check if the Personalisation Manager can successfully request predictions and preference outcomes from the personalisation sources based on changes in the context and service environment of the user.

### Preference Learning
The purpose of this test is to check if the C45 learning algorithm can learn preferences based on the user's action history.

### Preference Merging
The purpose of this test is to check if the resulting preferences by the C45 learning algorithm can successfully be merged with existing preferences.

### CAUI discovery
The purpose of this test is to check if the CAUI discovery can successfully discover a new CAUI model.

### CRIST discovery
The purpose of this test is to check if the CRIST discovery can successfully discover a new CRIST model.

### Community preference learning
The purpose of this test is to check if the community preference learning component can successfully learn community preferences.

### CACI Discovery (community intent)
The purpose of this test is to check if the CACI discovery component can successfully discover CACI model.

### CRIST Community Discovery
The purpose of this test is to check if the CRIST community discovery component can successfully discover a Community CRIST model.

### 4.1.6 User Agent

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Explicit Feedback</td>
<td>The purpose of this test is to check that the modal proposition pop-ups present the correct information to the user and return the correct ack/nack result to the feedback requestor.</td>
</tr>
<tr>
<td>Get Implicit Feedback</td>
<td>The purpose of this test is to check that the timed proposition pop-ups present the correct information to the user and return the correct ack/nack result to the feedback requestor.</td>
</tr>
<tr>
<td>Notify User</td>
<td>The purpose of this test is to check that the notification pop-ups present the correct information to the user.</td>
</tr>
<tr>
<td>Monitor User Action</td>
<td>The purpose of this test is to check that that User Action Monitor functionality can correctly handle user actions as they are received from services.</td>
</tr>
<tr>
<td>Share Resource</td>
<td>The purpose of this test is to check that the Resource Sharing functionality can correctly share a resource between multiple users depending on the sharing policy associated with the sharable resource.</td>
</tr>
<tr>
<td>Multi-agent collaboration</td>
<td>The purpose of this test is to check that agents can correctly collaborate (on behalf of the users) to identify mediated service configurations for shared services.</td>
</tr>
<tr>
<td>Apply personalised service adaptation</td>
<td>The purpose of this test is to check that the Decision Making and Conflict Resolution functionalities can decide upon and apply the appropriate personalised service adaptation.</td>
</tr>
</tbody>
</table>
### Multi-agent control

The purpose of this test is to check that the user can correctly control their user agents.

### 4.1.7 CIS Creation

This family of tests refers to the test cases where the user is performing the actions related to creation of a CIS (Directory, Mgt, LifeCycle, Privacy policies, etc).

This family should include test cases for the modules CIS Management, CIS Registry.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate a CIS</td>
<td>The purpose of this test is to check that the first CSS member of the CIS will activate this CIS.</td>
</tr>
<tr>
<td>Attach a privacy policy to the CIS</td>
<td>The purpose of this test is to check that a privacy policy can be attached to a CIS during its creation.</td>
</tr>
<tr>
<td>Create locally a CIS</td>
<td>The purpose of this test is to check if it is possible to create a CIS without worrying for now about registration parts. Is it possible to retrieve an identity from the Domain Authority? Is it created with the parameters filled by the user?</td>
</tr>
<tr>
<td>Deactivate a CIS</td>
<td>The purpose of this test is to check that the last CSS member of a CIS will deactivate it when living.</td>
</tr>
<tr>
<td>No CIS announced without privacy policy</td>
<td>The purpose of this test is to verify that it is impossible to announced a CIS (i.e. register it and set it as accessible for CSS) without a privacy policy attached to it.</td>
</tr>
<tr>
<td>Register a CIS</td>
<td>The purpose of this test is to check the registering of a CIS in a CIS Registry. Is it register? With good parameters?</td>
</tr>
</tbody>
</table>

### 4.1.8 Orchestrate a Community

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join a CIS</td>
<td>The purpose of this test is to determine whether a CSS is able to join a CIS</td>
</tr>
<tr>
<td>Leave a CIS</td>
<td>The purpose of this test is to determine whether a CSS is able to leave a CIS</td>
</tr>
<tr>
<td>Automatic Membership</td>
<td>The purpose of this test is to determine whether it is possible to automatically add and remove a CSS as member of a CIS based on rules, e.g. context, CSS profile or goal</td>
</tr>
<tr>
<td>Recommend CIS</td>
<td>The purpose of this test is to determine whether the system recommends a CIS to a user based on his preferences and context.</td>
</tr>
<tr>
<td>Suggest creation of a new CIS</td>
<td>The purpose of this test is to determine whether the system will suggest the creation of new CIS based on user activity/ context.</td>
</tr>
<tr>
<td>Equivalent CIS</td>
<td>The purpose of this test is to determine whether the system detects equivalent CIS, in order to avoid creating a new CIS when an equivalent one already exists.</td>
</tr>
</tbody>
</table>
Edit grouping rules | The purpose of this test is to determine whether it is possible to change the rules by which CIS memberships is determined.

Suggest CIS preferences | The purpose of this test is to determine whether the system can automatically suggest configuration parameters for a CIS

### 4.1.9 SN Connector

This set of test cases will include the testing of actions when the SOCIETIES platform is connecting to a Non-Federated Social Networks.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified Authentication</td>
<td>Test to check that the CSS member does not need to repeatedly log into each SN we have interfaces to.</td>
</tr>
<tr>
<td>Upload content</td>
<td>The purpose of this test is to check that we can share content (comment, link, file, etc) for a CIS to a SN through our connector.</td>
</tr>
<tr>
<td>View shared content</td>
<td>To test that uploaded content to a CIS which is in fact pushed to a SN is visible to another CSS through our Connector. The converse test is that non members of this CIS cannot view this shared content</td>
</tr>
<tr>
<td>Information extraction</td>
<td>Test our SN Connector to extract info about persons, groups based on interests similarities so that we can pervasively recommend groups to interested users</td>
</tr>
<tr>
<td>Single sign on Mechanism</td>
<td>To test the Identifier for a person or organisation such that there is no ambiguity and ensure we still have anonymity</td>
</tr>
</tbody>
</table>

### 4.1.10 Context

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community context extraction</td>
<td>The purpose of this test is to check the extraction of a context for a community. Different situation should be checked: permissions for context extraction, different types of data sources</td>
</tr>
<tr>
<td>Community context inheritance</td>
<td>The purpose of this test is to check extraction of community context from a parent community</td>
</tr>
<tr>
<td>Context events</td>
<td>The purpose of this test is to check the mechanism for providing context event to other components (publish/subscribe)</td>
</tr>
<tr>
<td>Context history update</td>
<td>The purpose of this test is to check that the context history is properly updated when changes in the context occur (new context events)</td>
</tr>
<tr>
<td>Context prediction</td>
<td>The purpose of this test is to check the the future context of a use can be predicted</td>
</tr>
<tr>
<td>Context refinement</td>
<td>The purpose of this test is to check the mechanism of context refinement based on the context gathered so far</td>
</tr>
<tr>
<td>Context similarity evaluation</td>
<td>The purpose of this test to check the algorithms for context similarity evaluation between two users</td>
</tr>
<tr>
<td>Context source registration</td>
<td>The purpose of this test is to check the registration of a new context</td>
</tr>
<tr>
<td>Source</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Context source update</td>
<td>The purpose of this test is to check the update of an already registered</td>
</tr>
<tr>
<td></td>
<td>context source</td>
</tr>
<tr>
<td>Individual context retrieval</td>
<td>The purpose of this test is to check the retrieval of the context for an</td>
</tr>
<tr>
<td></td>
<td>individual user</td>
</tr>
<tr>
<td>Individual context update</td>
<td>The purpose of this test is to check the update of the context for an</td>
</tr>
<tr>
<td></td>
<td>individual user</td>
</tr>
<tr>
<td>Individual context inference</td>
<td>The purpose of this test is to check the inference of the context for an</td>
</tr>
<tr>
<td></td>
<td>individual user</td>
</tr>
</tbody>
</table>

4.1.11 Privacy

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional remove of personal information in CIS</td>
<td>This test case on Privacy Manager and Data Obfuscation Manager is proposed to guarantee the remove of personal information and avoid the denial of erasing in CIS. The information eraser should be actuated by each of personal information access by CIS, the atomicity of access/erasing should be checked. Generally, the personal information seeker, e.g., CIS instance, should provide a proof of easier after access; and then Data Obfuscation Manager must check that proof.</td>
</tr>
<tr>
<td>CRUD (create, read, update and delete) of privacy policies</td>
<td>These test suits are proposed to check the basic data manipulations, including create, read, update and delete operations, towards a set of privacy policies. We mainly focus on the persistence and retrieval process in the privacy policy manager.</td>
</tr>
<tr>
<td>CRUD (create, read, update and delete) of privacy preferences</td>
<td>These test suits are proposed to check the basic data manipulations, including create, read, update and delete operations, towards a set of privacy preferences. We mainly focus on the persistence and retrieval process in the privacy preference manager.</td>
</tr>
<tr>
<td>Evaluation of privacy preferences</td>
<td>These test suits are proposed to check the correctness of privacy preference evaluation. We mainly focus on the evaluation results of a series of privacy preference change given by the privacy preference manager.</td>
</tr>
<tr>
<td>Trust Evidence Collecting</td>
<td>These test suits are proposed to check the whether the Trust Evidence Collector is able to capture all types of Trust Evidence.</td>
</tr>
<tr>
<td>Direct Trustworthiness and Reputation Estimation</td>
<td>These test suits are proposed to estimate the trustworthiness of the member staying in the same CIS. We should check whether the Trust Manager could correctly measure and estimate the reputation from trust evidence repository.</td>
</tr>
<tr>
<td>Intra-CIS Trusted Response</td>
<td>These test suits are proposed to check the digital signature of response between two CIS member. We should check whether the Trust Engine inserts the correct and corresponding digital signature into the content of response.</td>
</tr>
<tr>
<td>Maintaining (CRUD) of trust-related information</td>
<td>These test suits are proposed to check the base operations, including addition, retrieval update and deletion, towards trust evidences inside of trust repository.</td>
</tr>
</tbody>
</table>
These test suits are mainly proposed to evaluate the risk to accept a new member and to evaluate the risk to access to 3rd Party service or SNs. We should design the test units to evaluate the analytical results of privacy assessment by given varied privacy log, negotiation agreement and user perceived trust.

### 4.1.12 Trust

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct trust</td>
<td>Test if direct trust is calculated correctly. The calculation is done by Direct Trust Engine.</td>
</tr>
<tr>
<td>Indirect Trust</td>
<td>Test if indirect trust is calculated correctly. The calculation is done by Indirect Trust Engine.</td>
</tr>
<tr>
<td>Trust Repository</td>
<td>Test if Trust Repository is storing, updating and providing trust profiles correctly. When a few trust profiles are stored for various trusted entities, the profiles retrieved at a later point should be the same. Test also if profiles are updated correctly. This applies to both direct and indirect trust profiles.</td>
</tr>
<tr>
<td>User Perceived Trust</td>
<td>Test if user perceived trust is calculated correctly. The calculation is done by User Perceived Trust Engine.</td>
</tr>
</tbody>
</table>

### 4.2 Capability Tests

The goal of the capability test is to validate that all the observable external static capabilities of the Cloud Platform are valid with respect to the static conformance requirements expressed in the definition of the SUT.

### 4.3 Behaviour Tests

The goal of behaviour tests is to check the dynamic conformance of the SUT with respect to the dynamic conformance requirements specified in the specification of the SUT. Typically, this part will include tests on mobility.

### 4.4 Performance Tests

The goal of performance tests is to check the capacity of the SUT when it is submitted to a large number of events.

### 4.5 Scalability Tests

The goal is scalability tests is to check the capacity of the SUT when it is submitted to a large number of users.

### 4.6 Robustness Tests

The goal of robustness tests is to check the capacity of the SUT when it is submitted to stress tests.

### 4.7 Interoperability Tests

The goal of interoperability tests is to check the conformance of SUT towards the interoperability architecture.
5 Conclusion

The Deliverable D7.1 provides a first set of guidelines for integration and test of the SOCIETIES platform. The process includes different levels of integration and tests, each level is controlled by well-defined actors. Tools are also identified to perform integration and testing at the right level. Integration and testing phase is the most consuming task in terms of effort. The use of a remote Test Tool to guide the integration and testing process will provide the assurance of a good level of testing coverage by providing explicit objectives in terms of performances.

The next steps of task T7.1 are:

- Analyzing the list of components provided by the development work-packages (4,5)
- Performing a cross-check with the D8.2 requirements in terms of tests.
- Identifying the test purposes for all the test suites.
- Preparing the integration phase by implementing the test cases, the test tool (the specific libraries required to access the system under test), and the test scripts.

These above tasks shall be ready by in D7.4 (Final integration and test plan of the SOCIETIES system).

The integration and test phase will start in March 2012 and will run until July 2012 by executing the tests on the test bed. The results will be an integrated SOCIETIES Platform ready for User Trials.
6 References


