D6.1 DELIVERY MANAGEMENT PLAN AND TESTING SPECIFICATION

Description of the software development process

Project title	**Collaborative Recommendations and Adaptive Control for Personalised Energy Saving**
Project acronym	**enCOMPASS**
Project call	**EE-07-2016-2017 Behavioural change toward energy efficiency through ICT**
Work Package	**WP6**
Lead Partner	**SMOB**
Contributing Partner(s)	**PMI, EIPCM, SHF, NABU, WVT, SES, CERTH, NHRF, SMOB, KTU, GRA, PDX**
Security classification	**Public**
Contractual delivery date	**30/04/2017**
Actual delivery date	**28/04/2017**
Version	**1.0**
Reviewers	**PMI (C. Pasini, P. Fraternali), CERTH (S. Krinidis)**

History of changes

Version	Date	Comments	Main Authors
0.1	15/03/2017	DDP (Deliverable Development Plan) – definition of the document structure and the contributions expected from each partner	L.Caldararu
0.2	20/04/2017	Review and update on Architecture components	C.Pasini
0.3	24/04/2017	Review and update of the platform components	L.Caldararu
0.4	26/04/2017	Review and update of the release plan	L.Caldararu
0.5	26/04/2017	Review and update of requirements	M.Melenhorst
0.6	26/04/2017	Quality check	C.Pasini, P Fraternali
0.7	27/04/2017	Final review	P.Fraternali
0.8	27/04/2017	Final review	S.Krinidis
1.0	27/04/2017	Final version	L.Caldararu

Disclaimer

This document contains confidential information in the form of the enCOMPASS project findings, work and products and its use is strictly regulated by the enCOMPASS Consortium Agreement and by Contract no. 723059.

Neither the enCOMPASS Consortium nor any of its officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

The contents of this document are the sole responsibility of the enCOMPASS consortium and can in no way be taken to reflect the views of the European Union.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 723059.

TABLE OF CONTENTS

Executive Summary

This document is the Deliverable **D6.1: Delivery management plan and testing specification**, which, according to the Description of Work has the following goals:

D6.1 provides description of the software development process; code management tools; documentation to set up the development and testing environments.

This deliverable represents an outcome of task T6.1 Delivery Management Plan. Based on requirements that are being defined in T2.1 (User-centered requirements specification and design of behavioural interventions), it sets up a centralized and automatic build process verified by automatic unit tests. It enables an effective collaboration environment for developers and testers, who continuously are updated about the status of the whole development of enCOMPASS.

The main objective of this deliverable, is laying the ground for an effective collaboration environment for developers, who continuously are updated about the status of the whole development of enCOMPASS. The aim of delivery management is to lower deployment and integration risk by supporting error prone tasks (such as manual deployment) and install project automation tools.

It describes the technical standards and procedures, adopted in the enCOMPASS software development process; it also presents the plan for delivering enCOMPASS platform releases, as conceived at the current stage of the project (month 6).

- Chapter 1 resumes the objectives of the enCOMPASS project in terms of behavioural models that will be implemented in the software platform. Also, this chapter provides an early overview of the enCOMPASS platform components. The development strategy for enCOMPASS platform is evolutionary prototyping which means that it will go through several iterations following incremental refinements of the functional specification (Task 2.3) and architecture design (Task 6.2).
- Chapter 2 presents an overview of the planned requirements analysis methodology.
- Chapter 3 provides a description of the development process, methodology and tools.
- Chapter 4 describes the composition of enCOMPASS platform releases.
- Chapter 5 schedules the timetable for the planned functionality within each release.
- Chapter 6 provides the testing strategy, introducing the procedures that will be followed in the testing enCOMPASS components and artefacts.

1 INTRODUCTION

The enCOMPASS project develops an ICT platform for improving the management of energy demand thanks to the integrated use of **smart meters**, **sensors, social computation** based on advanced models of consumer behaviour.

The solution proposed by the enCOMPASS project will be able to:

- **Understand and model** the consumers' current behaviour on the basis of historical and real-time energy usage data;
- **Stimulate behavioural change** for energy saving with a holistic approach integrating innovative digital tools with smart home automation and a full-cycle model of sustained behavioural change;
- Make **energy usage data accessible to consumers** in a user-friendly, easy-to-understand way;
- Demonstrate that individual comfort levels can be maintained while achieving energy savings;
- Validate the **relative effectiveness of different types of behavioural change interventions** for different types of users, in different types of settings and in different climatic conditions;
- Make the enCOMPASS platform, digital tools, services and acquired energy data **available to designated third-parties** (in privacy-preserving ways) initiating the creation of a business ecosystem for the development and provision of value-added services for smart energy demand management;

From the technical viewpoint, building the enCOMPASS system is a challenge due to several factors stemming from the hybrid nature of the solution to be constructed; indeed enCOMPASS is:

- A **socio-technical system**, which must deliver an engaging user experience to attract and retain energy users.
- A **data-intensive** system, because it will acquire, integrate and process a vast amount of heterogeneous data.
- A **quasi-real time system**, because it will ingest metering at a high speed and volume.
- A **decision support system**, in its capacity to serve the data analytics and modeling needs of the utility managers.
- A **distributed system**, because it will be deployable also component-wise and in a Software as a Service model, which demands for a highly distributable and flexible architecture.

To tackle the development of a system of this nature, it is imperative to establish a principled development process, based on solid engineering standards.

The choice of enCOMPASS is to adopt an agile version of Model Driven Engineering [Ambler04], which conjugated the platform-independent nature of MDE with the lean development approach of agile processes.

The motivation of this approach is manifold:

- **Expertise of the Consortium**: the consortium partners are well acquainted with both MDE and agile methods.
- **Benefits in the project lifetime**: MDE delivers better documentation of the software (the models) and is amenable to a higher degree of automation (e.g., code generation) and is less prone to technology changes.

- **Benefits after the project conclusion**: models embody the technical knowledge in a more durable form than source code and thus facilitate post project exploitation, when novel scenarios may demand the adaptation of the enCOMPASS components to different technological platforms.

1.1 OVERVIEW OF THE ENCOMPASS ARCHITECTURE

As presented in the Project Description of Action (DoA) (see Figure 1), the original concept of the enCOMPASS architecture indicates the components and the processes that will be implemented in order to achieve individual and collective behavioural response to specific energy conservation policies.

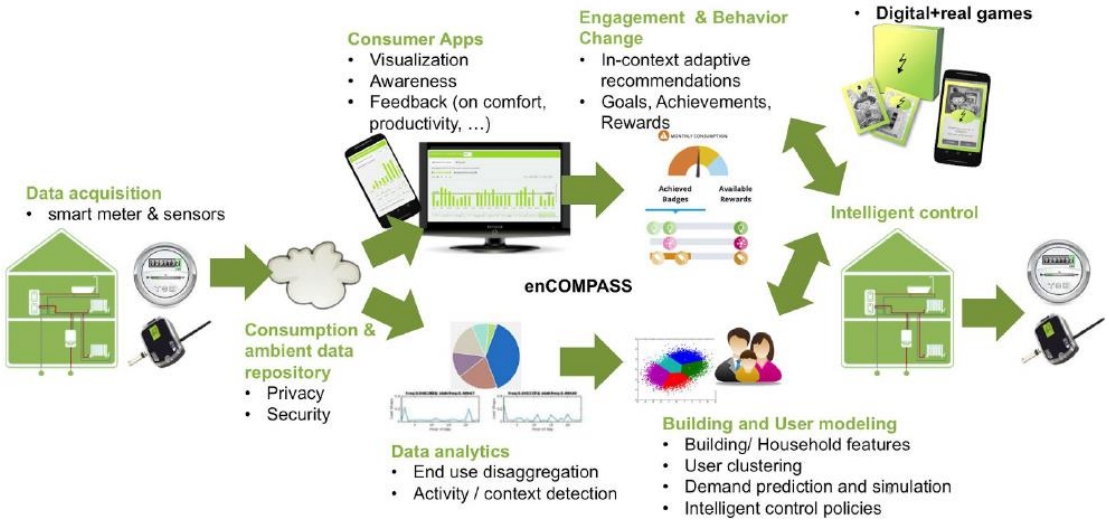

Figure 1: Overview of the enCOMPASS overall concept (DoA – enCOMPASS)

Seen as a system, enCOMPASS platform can be seen as a *negative feedback control system*.

A negative feedback control is specific to a system in which the output of the main process related to a proposed objective is fed back into the input with the purpose to reduce the effect of increasing the input. This kind of feedback control generally induces stability over a proposed objective.

In the real world of the enCOMPASS project, the purpose is a sustainable energy conservation policy, while the negative control feedback consists of inducing a shared understanding and motivation by the energy users, thus leading to a reduction in energy consumption, while not compromising the quality of life.

The main goal of the project is to ensure an efficient energy demand using social awareness, social gaming incentives and financial instruments. Following this social objective, enCOMPASS Platform architecture is designed with respect to the main data flows:

- **Input flow**: user behavioural data (usage metering, social game and social media profile).
- **Control flow**: social game incentives and price signals. This flow is supposed to trigger changes in user behaviour according to Energy Supplier objectives.

Besides the main data flows, the Platform must also accommodate utility subscriber profile data coming from Energy Supplier portals and reporting and analysis tools for Energy Supplier companies.

1.2 enCOMPASS FUNCTIONAL OBJECTIVES

The enCOMPASS Platform relies on **collecting data** from energy utilities, end-consumers gaming actions and social media, **processing data** using data analysis instruments such as gamification, then **measuring and exposing** user behaviour changes.

In designing the software platform, specific independent components can be identified, as well as their role of providing functionality as services to other components. Such components are independent of vendor, product or technology. This is the leading principle towards the decision of implementing *a Service-Oriented Architecture* for achieving enCOMPASS objectives.

1.2.1 Analysis and design methodology

The analysis and design methodology used for enCOMPASS Platform is Service Oriented. The theoretical basis and practical modelling tools reside mainly on:

- Service Oriented modelling and architecture (SOMA) from IBM.
- Service Oriented Architecture Modelling Language (SOAML) from OMG.
- Unified Modelling Language (UML) from OMG.
- Industry best practices and patterns for architecture and design.

1.3 LAYERS OF THE enCOMPASS ARCHITECTURE

Technical implementation of enCOMPASS Platform is based on a layered architecture. Each layer is designed with respect to separation of concerns principles. The proposed architecture is organized in four distinct layers:

- Data acquisition layer.
- Data/object layer.
- Business process layer.
- Consumer layer.

1.3.1 Data acquisition layer

This layer is responsible with bulk data acquisition and bulk data delivery. Inputs of this layer are:

- Energy usage data files from Utilities. Parallel processing of raw data files will be performed by open source Big data analysis platforms.
- Social media user data.
- Other REST based data sources, user portals of Energy Utilities.

This layer plays the role of a mediation component that handles raw data acquisition, transformation and storage in a format that can be used at upper layers.

1.3.2 Data/Object layer

This layer is responsible for data storage in SQL (and NoSQL formats where needed for efficient processing). This layer will expose services for upper level for basic access to data. It will store data such as:

- Energy usage data.
- Building general information and sensor data.
- User profile data.
- Game actions and rewards data.

- Exogenous data (e.g., meteo time series).

1.3.3 Business Process layer

This layer is responsible for implementation of business logic. This layer will expose business services for Consumer layer. Business level components are:

- **Gamification engine**. This component will provide game scenarios and will handle user interactions with the platform through social game clients. This is a platform built-in component, as it satisfies a critical enCOMPASS project objective.
- **Recommendation engine**: This component will provide personalized adaptive energy saving recommendations, based on the user's profile and context.

1.3.4 Consumer layer

This layer consists of client applications for services exposed by the Business Process Layer. Consumer of platform business services that are foreseen at this stage:

- Energy consumer applications (households, public building dwellers, schools).
- GWAP (Games with a Purpose) client application.
- Energy utility administrative application.
- Platform administration and configuration application.

As the platform implementation advances it will always possible to connect to the enCOMPASS services any client applications that implements the platform API specification.

1.4 enCOMPASS Components

At the moment of the writing of this deliverable, due in Month 6, various activities such as analysis meetings, conference calls, on-to-one discussions between partners are being undertaken in order to define the enCOMPASS platform architecture. The definitive enCOMPASS platform architecture will be provided in deliverable D6.2 Platform architecture and design, due in Month 12.

The components foreseen at this stage for building the enCOMPASS platform are the following ones:

- The **Smart meter and sensor data management component** [SMOB] deals with the acquisition of data streams from smart meter and their consolidation within the enCOMPASS database.
- **Behavioural change apps** [PMI]: web and mobile applications developed to visualize energy usage, deliver recommendations and engage the user with serious game mechanics and gamification techniques.
- The **Portal data exchange component** [SMOB] deals with the communication between the enCOMPASS platform and a third party application already supporting the interaction with the various types of users. Such application may comprise a customers' portal of the utility company, or a B2E application for managers and operators.
- The **Social data exchange component** [PMI] deals with the communication between the enCOMPASS platform and social network communities where the utility company has a presence or consumers are already enrolled. Such communication may serve the purpose of advertising energy awareness initiatives, disseminate the social games, or publishing customers' achievements.
- The **Gamification engine component** [PMI] embodies rules for transforming users' actions into gamification scores and achievements.
- The **Gamification Engine Admin Portal** [PMI] allows one to configure gamification objects like actions, badges and rewards and includes tools to monitor customer gamification activities.

- The **Energy Game Digital Extension** [PMI-KAL] consists of mobile apps (typically, mobile digital games) targeted at energy consumers for letting them have a playful experience while increase their individual and social awareness about sustainable energy consumption behaviour. Such component provides activities logs for the gamification engine to compute the user's gamification achievements and scores.
- **Data analysis and user modelling** [CERTH]: algorithms for extracting activity from sensor and app data, profiling different types of user behaviour, inferring activity context, predicting reactions to stimuli (e.g., saving tips).
- **Data analysis and building modelling** [CERTH]: algorithms for estimating the comfort levels and energy behaviour of buildings based on a reduced set of measured input parameters dependent on the class of the buildings. They allow projecting the impact of consumers' actions on energy consumption and on the status of the building, to complete the user context. The base model includes: building characteristics (i.e., space allocation and size, neighbouring spaces, building size, location of the energy consumption of subsystems, etc.), user profile, local climate data, energy consumption of subsystems (i.e., HVAC, plug loads etc.).
- **Adaptive in-context recommendation engine** [GRA]: used in order to map user's models and behavioural data into personalized recommendations for energy saving. Users will be classified in categories and activity patterns will be extracted from consumption, sensors' and psychographic data (in a privacy respecting way), in order to compute saving tips that fit the current context of the specific user.

The full technical details of the enCOMPASS platform and application design will be provided in the deliverable: D6.2 Platform Architecture and Design, which is due at month 12.

2 SPECIFYING REQUIREMENTS IN THE ENCOMPASS PROJECT

enCOMPASS will develop a socio-technical system, which highly depends on the human factors in the interaction with the technology. Therefore a suitable emphasis is placed in the methodology for collecting requirements, which must be user-centred and give appropriate attention to the usability and social acceptability factors in application design.

To emphasize this aspect of development, we anticipate in this section the view of the planned requirements analysis methodology and the resulting requirements specification format that will be employed in the enCOMPASS project. The fine-grained requirements approach and early use cases will be defined in D2.1 Use cases and early requirements.

2.1 USER-CENTERED DESIGN METHODOLOGY

The requirements analysis methodology follows the iterative human-centred design process defined in ISO 13497 [ISO99]. To understand user needs following this process, first the target group and context of use are defined, followed by the specifications of user requirements, the development of design solutions and the evaluation of these solutions with users and stakeholders (see Figure 2). The results feed into the next iteration cycle, in which the activities are repeated until the solution is considered mature enough. During these multiple iterations, feedback is obtained both from end-users (user pull) and technical partners (technology push), which aims to construct requirements that are both technically feasible and grounded in user-needs.

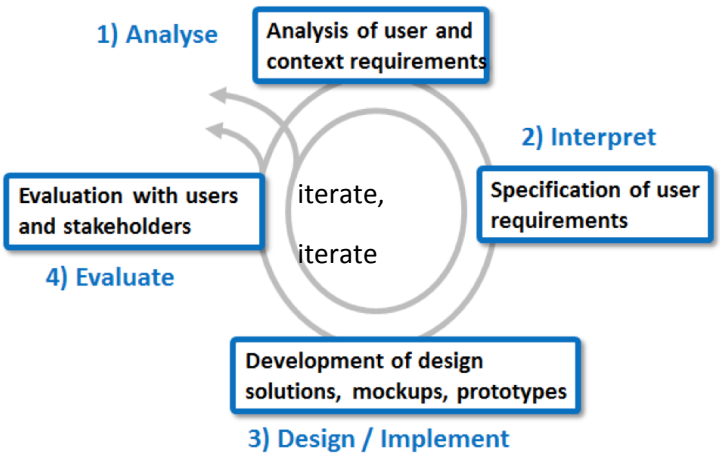

Figure 2: Human-centered design process according to [ISO 13407]

2.2 REQUIREMENTS SPECIFICATION MODEL

Based on the requirements analysis activities, the requirements for the enCOMPASS applications will be specified in the following way. High-level user stories (scenarios of use, [Carrol95]) will be defined to describe what a set of users do and experience as they perform a set of specific tasks in a specific context. The user stories will be described by a short structured narrative. Based on the users stories, use cases will be defined. Additionally, visual user interface mock-up will be created, illustrating the given functionalities from the user perspective. The mock-ups will be used to elicit user feedback and to align technical feasibility.

For each use case, a list of functional and non-functional requirements will be defined. These will be formalized using one of the well-known use case templates (e.g. [Cockburn01]). This basic requirements specification model is visualized in Figure 3. The details will be defined in D2.1 Use cases and early requirements.

The requirements following the described specification model will be delivered within the deliverables D2.1 Use cases and early requirements, D2.2 Final requirements and D2.3 Functional specification.

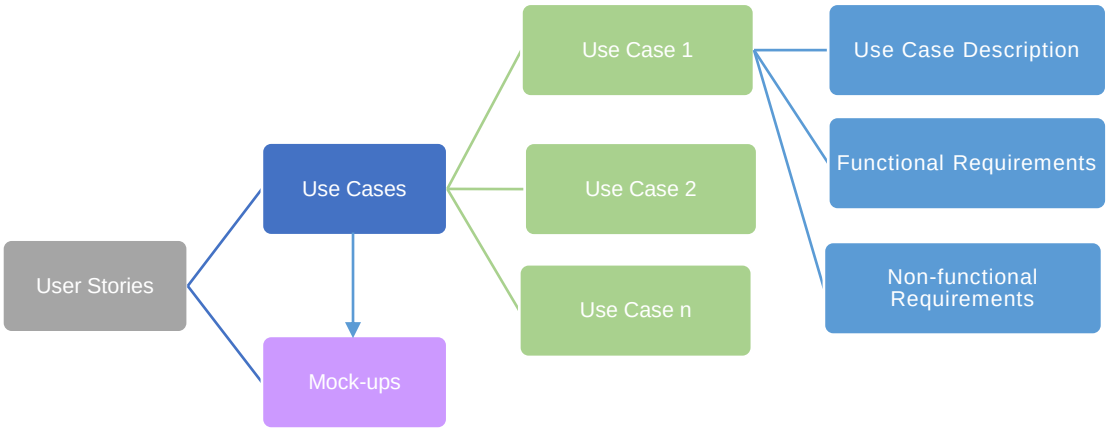

Figure 3: User-centered requirements specification model in enCOMPASS

The results of the application of the employed requirements analysis and specification methodology are collected in the deliverables:

- D2.1 Use cases and early requirements, which is due at month 8.
- D2.2 Final requirements, which is due at month 12.

3 DEVELOPMENT PROCESS AND METHODOLOGIES

In order to reach the project objectives, a proper technical management of the enCOMPASS platform implementation must be conducted. For selecting the right development methodology critical aspects as the following ones have to be considered:

- The usage of open standards for developing and distributing the enCOMPASS platform.
- The identification of highly reusable independent components in the designed software system.
- The strength of the relationship between the software components and between the enCOMPASS system and the external integrated applications.

As a result of an extensive analysis based but not limited to the above aspects, the methodologies selected for developing and integrating the software components of the enCOMPASS Platform is **Model-Driven, Agile** and **Service Oriented.**

The theoretical basis and practical modelling tools and patterns of this Architecture document reside mainly but not limited to:

- Object oriented system modelling with the Unified Modelling Language (UML) from OMG [OMG-UML].
- Interactive system modelling with the Interaction Flow Modelling Language [OMG-IFML].
- Data Modelling with the Entity-Relationship Model [Chen76].
- Service-oriented modelling and architecture (SOMA) with IBM [IBM-SOA] and the Service Oriented Architecture Modelling Language (SOAML) from OMG [OMG-SOAML]

3.1 MODEL DRIVEN DEVELOPMENT PROCESS

enCOMPASS will develop a platform consisting of a backend for data acquisition, storage, processing and the behavioural modelling of users, coupled to a set of graphical user interfaces, ranging from gaming interfaces to business portals to data analytics dashboards, offered to a variety of stakeholder.

The capabilities that enCOMPASS must deliver to the users span such aspects as information browsing and analytics, hypertext-style navigation, form-based interaction, and interface personalization, both in consumer applications and in business information systems. Such functionalities must be implemented on top of a variety of devices, technological platforms, and communication channels.

To address this complexity, enCOMPASS will exploit software development approaches based on a **Platform Independent Model (PIM),** which can be used to express the software design decisions independently of the implementation platform, according to the so-called Model Driven Engineering paradigm (MDE) [BCW12].

Specifically, enCOMPASS development will follow the MDE incarnation proposed by the **Object Management Group Model Driven Architecture (MDA)** and, more in general, which is a well-known international body of standards for the MDE development approach.

Furthermore, the development of complex and heterogeneous applications such as the enCOMPASS back-end platform and GUIs will be addressed with agile approaches, which traverse several cycles of “problem discovery” / “design refinement” / “implementation”. An iteration of the development process generates a prototype or a partial version of the system. Such an incremental lifecycle is particularly appropriate for modern Web and mobile applications, which must be deployed quickly and change frequently during their lifetime to adapt to the user’s requirements. Figure 4 schematizes the development process adopted in

enCOMPASS platform and positions the various modelling notations and standards exploited in the project within the flow of activities.

Requirements specification collects and formalizes the information about the application domain and expected functions. The input is the set of business/user requirements that motivate the application development, and all the available information on the technical, organizational, and managerial context. The output is a functional specifications document comprising:

- The identification of the user roles and of the use cases associated with each role.
- A data dictionary of the essential domain concepts and of their semantic relationships.
- The workflow embodied in each use case, which shows how the main actors (the user, the application, and possibly external services) interact during the execution of the use case.

In addition, non-functional requirements must also be specified, which include performance, scalability, availability, security, and maintainability. When the application is directed to the general public, e.g., the energy users, requirements about the "look & feel" and usability of the interfaces assume special prominence among the non-functional requirements. User-centred design practices can be applied, which rely on the construction of realistic mock-ups of the application functionality, which can be used for the early validation of the interface concepts and then expanded into more detailed and technical specifications during the front-end modelling phase. The user-centred methodology explained in Section 3 will drive the activities in the Requirements specification task.

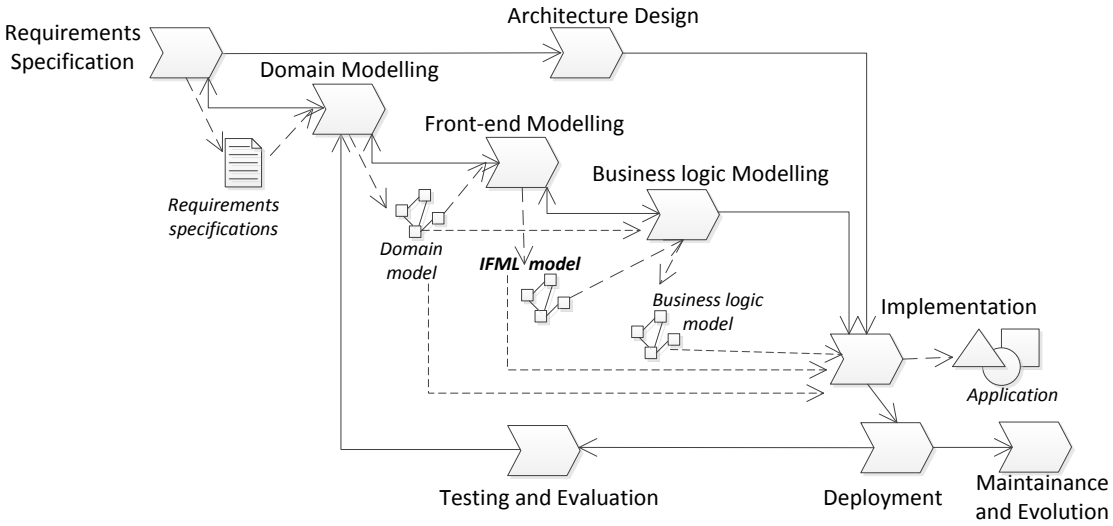

Figure 4: Role of IFML in the development process of an interactive application

- **Domain modelling**[1] organizes the main information objects identified during requirements specification into a comprehensive and coherent Domain Model. Domain modelling is a well-established discipline: the first conceptual data modelling language, the Entity-Relationship model, was proposed in 1976, and ever since new modelling languages have been proposed, including UML. Correspondingly, modelling practices and guidelines have been consolidated; in particular, domain modelling for interactive applications exploits suitable design patterns, discussed in chapter 3. The entities and associations of the Domain Model identified during domain modelling are referenced in the front-end design models, to describe what pieces of data are published in the interface.

1 "Domain Modelling" is the locution normally employed in object-oriented methodologies, whereas conceptual database design normally refers to "Data Modelling".

- **Front-end modelling** maps the information delivery and data manipulation functionality dictated by the requirements use cases into a front-end model. Front-end modelling operates at the conceptual level, and is where Interaction Flow Modelling Language (IFML) [OMG-IFML] comes into play. The designer may use IFML to specify the organization of the front-end in one or more top-level view containers, the internal structure of each view container in terms of sub-containers, the view components that form the content of each view container, the events exposed by the view containers and components, and how such events trigger business actions and update the interface.
- **Business logic modelling** specifies the business objects and the methods necessary to support the identified use cases. UML static and dynamic diagrams are normally employed to highlight the interfaces of objects and the flow of messages. Process-oriented notations, such as UML activity and sequence diagrams, BPMN process models, and BPEL service orchestrations provide a convenient way to represent the workflow across objects and services. The actions specified in the business logic design can be referenced in the front-end model, to show which operations can be triggered by interacting with the interface.
- Data, front-end, and business logic design are interdependent activities executed iteratively; the precedence order of Figure 4 is only illustrative: in some usage scenarios, work could start from the design of the front end and the data objects and actions could be discovered a posteriori by analysing what information is published in the interface and what operations are requested to support the interactions.
- **Architecture design** is the definition of the hardware, network and software components that make up the architecture on which the application delivers its services to users. The goal of architecture design is to find the mix of these components that best meets the application requirements in terms of performance, security, availability and scalability, and at the same time respects the technical and economic constraints of the project. The inputs of architecture design are the non-functional requirements and the constraints identified during business requirements collection and formalized in the requirements specifications. The output may be any specification that addresses the topology of the architecture in terms of processors, processes and connections, such as UML deployment diagrams.
- **Implementation** is the activity of producing the software modules that transform the data, business logic, and interface design into an application running on the selected architecture. Data implementation maps the Domain Model onto one or more data sources, by associating the conceptual-level constructs to the logical data structures (e.g., entities and relationships to relational tables). **Business logic implementation** creates the software components needed to support the identified use cases; the implementation of individual components may benefit from the adoption of software frameworks, which organize the way in which fine-grain components are orchestrated and assembled into larger / more reusable functional units and also cater for non-functional requirements, like performance, scalability, security, and availability. Business logic may also reside in external services, in which case implementation must address the orchestration of calls to remote components such as Web APIs. **Interface implementation** translates the conceptual-level Containers and Components into the proper constructs in the selected implementation platform. View Containers may interoperate with business objects, deployed either in the client layer or in the server layer.
- **Testing and evaluation** verify the conformance of the implemented application to the functional and non-functional requirements. The most relevant concerns for interactive applications testing are:

- Functional testing: the application behaviour is verified with respect to the functional requirements. Functional testing can be broken down into the classical activities of module testing, integration testing and system testing.
- Usability testing: the non-functional requirements of ease of use, communication effectiveness, and adherence to consolidated usability standards are verified against the produced front-end.
- Performance testing: the throughput and response time of the application must be evaluated in average and peak workload conditions. In case of inadequate level of service, the deployment architecture, including the external services, must be monitored and analyzed for identifying and removing bottlenecks.

- **Deployment** is the activity of installing the developed modules on top of the selected architecture. Deployment involves the data layer and the software gateways to the external services, and the business and presentation layer, where the interface modules and the business objects must be installed.
- **Maintenance and evolution** encompass all the modifications applied after the application has been deployed in the production environment. Differently from the other phases of development, maintenance and evolution are applied to an existing system, which includes both the running application and its related documentation.

Model Driven Engineering has important implications not only during the production of software but also for other development activities.

- Implementation may exploit model transformations and code generation to produce prototypes of the user interface or even the fully functional code.
- Testing and evaluation can be anticipated and performed on the software models, rather than on the final code. Model checking may discover inconsistencies in the design of the front-end (e.g., unreachable statuses of the interface) and suggest ways for refactoring the user interface for better usability (e.g., recommend uniform design patterns for the different types of user interactions, such as searching, browsing, creating, modifying, and deleting objects).
- Finally, maintenance and evolution benefit most from the existence of a conceptual model of the application. Requests for changes are analyzed and turned into changes at the design-level. Then, changes at the conceptual level are propagated to the implementation, possibly with the help of model-to-code transformation rules. This approach smoothly incorporates change management into the mainstream production lifecycle, and greatly reduces the risk of breaking the software engineering process due to the application of changes solely at the implementation level.

3.2 AGILE DEVELOPMENT PROCESS

Model-Driven Development can be used in conjunction with agile methodologies, to attain a development process called **agile model-driven** [Ambler04].

enCOMPASS will adopt **an agile approach towards model-driven development and software integration** for delivering the platform. Based on the preliminary technical scenarios identified at this stage of the project, the applications and the services that will constitute the software platform will be developed as an iterative process with permanent feedback from the user communities and energy utilities. Adopting an agile approach will ensure a proper way to react quickly and to respond accurately to the changes that are inevitable during the development process.

Scrum is a lightweight project management process [Scrum Methodology, cPrime] that can manage and control software and product development. Instead of promoting the traditional analysis, design, code,

test, deploy "waterfall" approach. Scrum proposes iterative and incremental practices. Similarly, instead of being "artefact-driven", whereby large requirements documents, analysis specifications, design documents are created, Scrum requires fewer artefacts in order to start working. Concerning the software development, Scrum concentrates on writing software that produces business value.

Scrum allows working on small pieces at a time, in an iterative approach. Each iteration consists of some requirements gathering, some analysis, some design, some development and some testing culminating in an iterative release cycle with many deployments.

Scrum Roles

Scrum uses three “roles": Product Owner, Scrum Master and Project Team.

- The Product Owner is possibly a Product Manager or Project Sponsor, a member of Marketing or an Internal Customer.
- The Scrum Master is key person "represents management to the project". Such a role usually filled by a Project Manager or Team Leader. They are responsible for enacting Scrum values and practices. Their main job is to remove impediments, i.e. project issues that might slow down or stop activity that moves the project forward.
- The Project Team usually consists of between 3 to 10 members. The team itself is cross-functional, involving individuals from a multitude of disciplines: QA, Programmers, Testers, UI Designers.

The Process

"Out of the box" Scrum is described by Figure 5. Most projects have a list of requirements (type of system, planning items, type of application, development environment, user considerations, etc.) Scrum records requirements in a Product Backlog. Requirements need not be precise nor do they need to be described fully. As with most projects, the requirements are sourced from the expected users or "the business". The Product Owner prioritizes the Product Backlog: items of importance to the project/business, i.e. those items that add immediate and significant business value, are bubbled up to the top.

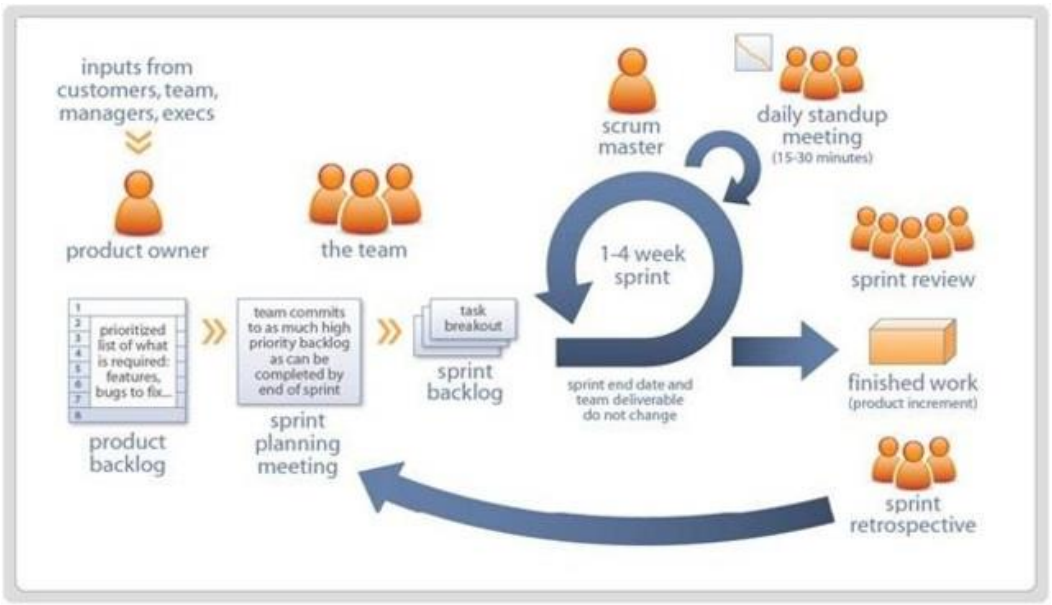

Figure 5: Scrum process

The Project Team responsible for doing the actual work then creates a Sprint Backlog: this comprises of Product Backlog items that they believe can be completed within a 30 day period. The Project Team may liaise with the Product Owner and others in order to expand item(s) on the Sprint Backlog. After 30 days have elapsed, the team should have a "potentially shippable product increment".

The Product Owner, the Scrum Master and the Project Team will make an initial pass over the Product Backlog items where they work out roughly how long each item will take. Initially, these are estimates, best guesses. As time progresses, they will find out if the estimate was even close.

Scrum allows refining the initial estimates on-the-fly: if a task will take longer than envisaged, it offers the ability to say so before the tasks starts. By only ever working with small work packages (time-boxed to 30 days), any schedule/requirement issues are dealt with as soon as they are identified, not much further downstream where the cost of recovery is considerably higher.

3.3 REFERENCE ARCHITECTURAL PATTERNS

The overview of the enCOMPASS platform emphasize a system architecture composed by decoupled components and packages of components that are orchestrated to work together in order to produce the expected outcome. The key principle used in designing and developing enCOMPASS platform is the “separation of concerns” – that is separating the platform into distinct sections, such that each section addresses a separate concern (software element). The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently.

Further, close to the development side, the design principles are implemented through design patterns. These are critical for underlining a correct approach in designing and writing maintainable and reusable code. A design pattern is a reusable solution that can be applied to commonly occurring problems in software design. Selecting the right design pattern that will be used for implementing the software project is a strategic choice for the success of the enCOMPASS project in terms of: development, adaptability to requests for change, usability from both user side and utility side and overall achievements.

3.3.1 Design Patterns

Model-View-Controller (**MVC**) is a design pattern that enforces the separation between the input, processing, and output of an application. To this end, an application is divided into three core components: the model, the view, and the controller. Each of these components handles a discreet set of tasks.

The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. The **controller** interprets the input events (input from mouse and keyboard) from the user, commanding the model and/or the view to change as appropriate. Finally, the **model** manages the behaviour and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). Figure 6 shows a graphical representation of the MVC paradigm.

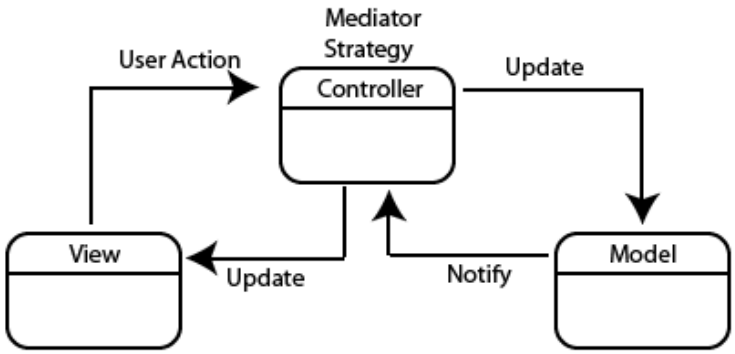

Figure 6: Model-View-Controller paradigm

3.3.2 Three-tier architecture

From a conceptual point of view, the MVC paradigm reflects the separation of the tiers within the enCOMPASS platform **multi-tier** architecture.

By segregating the applications into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. The **Three-tier** architecture is typically composed of a presentation tier (implementing the user interface), a domain logic tier (implementing the business rules), and a data storage tier (implementing data access) that are developed and maintained as independent modules. The three-tier architecture is intended to allow any of the tiers to be upgraded or even replaced independently, in response to changes in requirements or technology. For example, changing the operating system in the presentation tier would only affect the user interface code. Also, using a different DBMS than the platform original (e.g. PostgreSql instead of MySql), will only affect the data storage tier.

Figure 7 shows a sample graphical representation of the three-tier architecture concept implemented within a use-case of the enCOMPASS platform. The objective of the application use-case is to display a chart representing the energy consumption of a family for the current month.

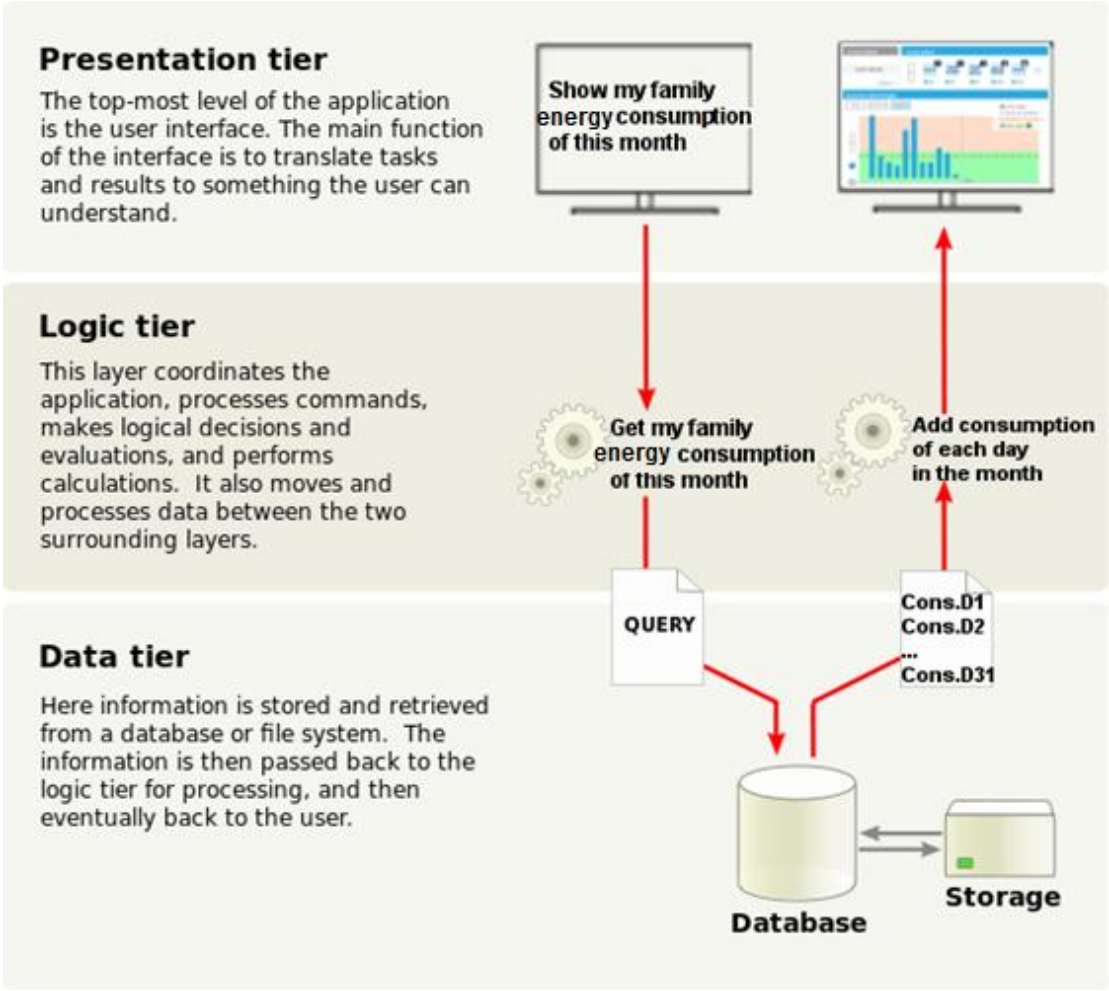

Figure 7: Example of Three-tier architecture representation of a enCOMPASS use-case

3.3.3 Application frameworks

The key objectives of the enCOMPASS platform - that are education in energy usage and openness in accepting changing of behaviour – are permanently reflected in the choices for implementation of the platform, including the approaches towards developing the software side. The enCOMPASS platform is entirely based on **Open Source software** with **Java** as central technological choice while the code of the specific components that will be developed for fulfilling platform specific objectives will be made public for each project release, also.

The standard structure of the applications will base on **software frameworks**.

By this approach, the applications are in full compliance with the business rules, that is structured and that is both maintainable and upgradable. Also, the development process become faster, because it allows developers to save time by re-using generic modules in order to focus on creative areas.

As an example, a software framework will keep the developer from having to spend days in order to create an authentication form - which is a repetitive task in man projects. The time that is saved can be dedicated to more specific components as well as to the corresponding unit tests, thus providing solid, sustainable and high quality code.

As a result of understanding the preliminary requirements for development, the process of software development foresees the usage of some of the most efficient open source programming frameworks and best practices in Java.

Hibernate [HIB] and **Spring** [SPRING] are open-source Java frameworks that simplify developing and integrating Java/JEE applications. **Hibernate** framework will be used for solving the requirements of managing and persisting data to the platform database. **Spring** framework provides a multitude of services spread over the application architecture, such as inversion of control, aspect-oriented programming, modularizing common behaviours, decoupling the application components as well as integrating components.

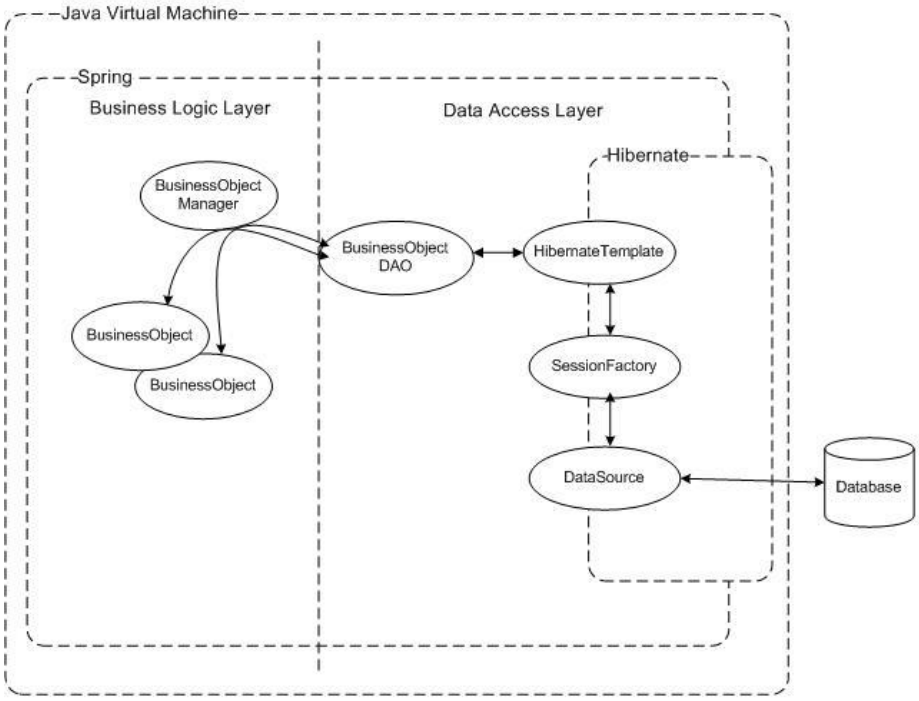

Figure 8: Conceptual integration of Hibernate and Spring in a Java application

3.4 CODING CONVENTIONS AND GUIDELINES

Software production from enCOMPASS partners is expected to follow not only common design and architectural patterns, but also common rules in naming convention, quality certification, development systems, libraries, documentation and forms of Software delivery: all these issues are taken in consideration in the following, in order to set up proper foundations for the Software building. Where available at the time this document is issued, examples of guidelines and system configurations are provided as samples.

According to this goal, this section provides specific procedures for developers who have to deliver enCOMPASS components. These procedures are intended to ensure that delivered components are installable and work properly. Ideally, by using these procedures, components need only to be retrieved from binary repository, installed, configured and tested to run, optimising the overall production cycle lead time.

These paragraphs are intended to give some common conventions for enCOMPASS software development. In the case of enCOMPASS, for all kinds of artefacts produced, (components, pipelets and pipelines) the development conventions should be adopted.

3.4.1 Naming Conventions

The definition of a common and agreed-upon naming convention for developed bundles and created packages is basic in a project with various partners.

- The names chosen should easy, giving an idea about the feature improved and the kind of bundle produced.
- The name of the project should always be present in all features and bundles.
- In the name of packages could appear also the coded name of the partner.

e.g.:

Feature Name: enCOMPASSproject.process.smob.wikilyrics

Base package name: eu.enCOMPASS.process.smob.wilkilyrics

3.4.2 Guidelines Writing Source Code

The source code should adopt the standard common java and JavaBean standard rules:

a. The package names should have only small letters;
b. All the class names should start with a capital letter;
c. All the class attributes and member names and also local variable names should start with a small letter;
d. When a class, an attribute, a member or a variable name is composed of more than a word, the letter of each word should be capital apart from first one;
e. The public static constant should be in capital letters and if more words, tied by an underscore;
f. All the class attributes should be private;
g. All the attributes must be used with their own getters and setters;
h. Getter and setter methods should be public;
i. Getter methods don’t have parameters and must return value type of same set methods;
j. Getter methods that don’t return boolean values must start with “get”;
k. Getter methods that return boolean values should start either with “is” or “get”;
l. Adding list methods should start with “add”;
m. Removing list methods should start with “remove”.

3.4.3 Exception Handling

Proper exception handling management is really important in framework-based software like in enCOMPASS.

It is important that the checked exceptions are well thrown and well managed without changing their meaning. If a kind of exception is thrown, the calling code should not manage that, should not change the message but at least should create a new exception based on the original one.

Unchecked exception must be well prevented using right checks on parameters, variables and attributes. Each parameter should be checked in its integrity to prevent null pointer exception, format exception and arithmetical exceptions.

3.4.4 Logging Guidelines

In a highly distributed system like enCOMPASS it is really important to create a useful and well-designed logging system. There are some easy rules to follow.

Avoid a static reference to an apache commons log instance:

e.g.: It is good doing

```
private final Log _log = LogFactory.getLog(MyClass.class);
```

Always check log level before logging:

e.g.: It is good doing

```
if (_log.isErrorEnabled()) {
   _log.error("Your error message", e);
}
```

Don't log an exception before throwing it or issuing a new one:

e.g.: It is bad doing

```
...
if( paramXY == null ) {
   if (_log.isErrorEnabled()) {
      _log.error("paramXY is not set");
   }
   throw new NullPointerException("paramXY is not set");
}
```

3.4.5 Third Party Components Integration Guidelines

The installation of third party libraries (libraries, components and other software artefacts not developed within enCOMPASS project – for example some graphical interfaces may use third party libraries and be preliminary to the installation of enCOMPASS components. These dependencies should be clearly reported in the ***installation and configuration file***.

Libraries should be delivered jointly with related components where possible. If this is not possible because of technical or legal (licenses) or other reasons, the installation and configuration file should report download locations, versions and describe installation and configuration details with reference to the proper URLs. All this information is collected in the ***installation and configuration file***.

3.5 CONTINUOUS INTEGRATION

Performing a high quality software development process that delivers trustful results is a key factor in reaching the project objectives. This must-have condition has to be fulfilled while meeting real-life constraints such as developing collaboration projects in a loosely coupling environment. This is a kind of condition that it is unavoidable when organizations of different types, shapes and sizes are working together at collaboration projects as well as when the project itself aims to integrate other external projects.

These are reasons why an advanced and intelligently adapted practice is needed during enCOMPASS platform development time. **Continuous integration** (**CI**) is the practice of merging all developer working copies with a shared mainline as often as possible. Each check-in is then verified (tested, compiled, installed and audited) by an automated build, allowing teams to detect problems as early as possible. Because **CI** requires integrating so frequently, there is significantly less back-tracking to discover where things went wrong, so that the developers can spend more time for building features than for debugging errors.

CI is backed by several important principles that ensures a high quality software development process for the enCOMPASS platform. Such principles are:

- Maintaining a single source repository.
- Automating the build.
- Making the builds self-testing.
- Keeping the build fast.
- Testing in a clone of the production environment.
- Making it easy for anyone to get the latest executable.
- Making it possible for everyone to see what's happening.
- Automating the deployment.

The development team prepared the development environment by installing and configuring the tools for **Continuous integration** (**CI**) [ThoughtWorks].

The **CI** environment uses **Jenkins** [Jenkins] as continuous integration open-source server that receives by automated token the new code that was committed to the **Bitbucket** [Bitbucket] (Git based) software repository that will accommodate all open source repositories. **Jenkins** then tests, packages and submits the code to **Sonar** [Sonar] automatic code reviewer. After successfully passing the code audit the element is saved or released in **Nexus** [Nexus] – the repository manager for artefacts. The outcome of each iteration is reported by e-mail.

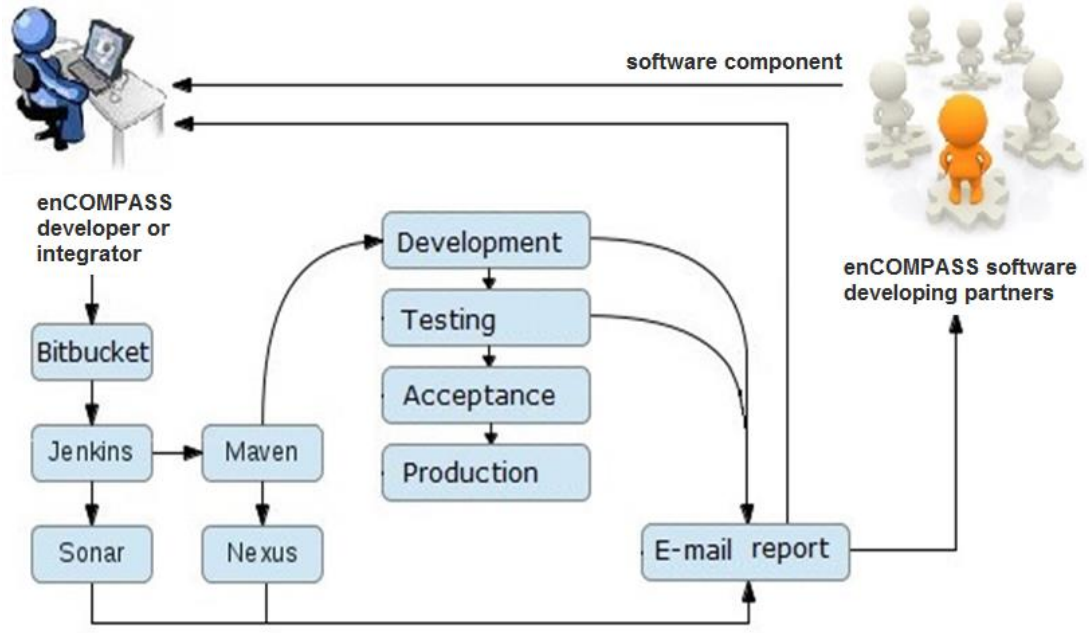

Figure 9: Continuous integration process

3.5.1 Continuous integration flow description

Bitbucket is a web-based hosting service for projects that use Git revision control and source code management system. Bitbucket is used to host the code and also as an issue tracker.

The code submitted to Bitbucket by the software developers is then automatically sent to **Jenkins** for proper testing. Jenkins is an open-source continuous integration server. It is designed to test and report on isolated changes in a larger code base in real-time. Jenkins is **a single point of entry** for building, testing, packaging and static analysis for the modules of the enCOMPASS platform.

In the software development configuration of the enCOMPASS platform, Jenkins builds a project using **Maven** [Maven]. Maven is a project building, management and comprehension tool based on the POM (Project Object Model). After building the project and testing using Maven the project is then sent to **Sonar** for a proper code analysis, review and audit. Sonar is a code quality platform it covers – coding rules, potential bugs, complexity, and duplication. Sonar issues a report that is sent to the parties interested according to the configuration.

After the code has been tested by Sonar, Jenkins then sends the code to the **Nexus** artefact repository manager for saving or releasing. **Selenium** [Selenium] is a suite of tools for automating web browsers across many platforms for testing purposes. It will be used as a build step in Jenkins in order to test the enCOMPASS platform and components.

4 WHAT MAKES AN ENCOMPASS RELEASE

Each enCOMPASS platform release is structured in artefacts. A release artefact is a tangible by-product produced during the software development process. The release artefact is a static object, which will not change in the project repository. Released artefacts are considered to be stable in order to guarantee that builds, which depend upon them, are solid and repeatable over time.

Each component of the enCOMPASS platform has a specific structure, depending on the expected outcome. Generically, each component consists of:

1. External Interfaces (Web Applications).
2. Business services.
3. Protocols.
4. Integration services.

According to this structure, each enCOMPASS release will be composed by group of artefacts corresponding to the advancement achieved at the specific time, provided in packaged way. Moreover the artefacts will be delivered together with the release notes document.

For example, a web application will be released as a WAR file artefact. The WAR file will be associated with a PGP signature, an MD5 and SHA checksum that can be used to verify both the authenticity and integrity of the binary software artefact. The code repository will include a set of descriptive attributes: groupId, artefactId, version, and packaging.

4.1.1 Group Identifier (groupId)

A group identifier groups a set of artefacts into a logical group. Groups are often designed to reflect the organization under which a particular software component is being produced. For example, software components being produced by the company SETMOB member of the enCOMPASS project Consortium are available under the groupId *org.enCOMPASS.smob*.

4.1.2 Artefact Identifier (artefactId)

An artefact will have an identifier for a software component. An artefact can represent an application or a library; for example, when creating a simple web application the project might have the artefactId “simple-webapp”, while when creating a simple library, the artefact might be “simple-library”. The combination of groupId and artefactId must be unique for a project.

4.1.3 Version (version code)

The version of a project follows the established convention of Major, Minor, and Point release versions. For example, a simple-library artefact has a Major release version of 1, a minor release version of 2, and point release version of 3, the version would be 1.2.3. During the development, versions can also have alphanumeric qualifiers that are often used to denote release status. An example of such a qualifier would be a version like “1.2.3-BETA” where BETA signals a stage of testing meaningful to consumers of a software component.

4.1.4 Packaging (packaging)

Packages describe any binary software format including but not limited to JAR, WAR, ZIP, SWC, EAR, SAR.

5 RELEASE PLAN

In respect of the initial work plan schedule identified in the DoA, the release scheduling details the functionalities expected in each platform prototype. The enCOMPASS project timeline comprises three main development phases. The end of each phase corresponded to a enCOMPASS release, which collects the progresses – in terms of artefacts- achieved at that time. Table 1 recollects the initial planning of release content, from the DoA.

Delivery Date	Release version	Objective
M18	1.0	The first prototype spans over the first 18 months of the project. It will provide the implementation of the first use cases defined for the pilots, based on user and system requirements, system architecture and data models specifications (**M1- M12**). The core integration will take place, leading to the 1st prototype (Release R1, **delivered at M18**), which exploits available components adapted to the requirements of enCOMPASS. R1 will incorporate features shown in Table3. Evaluation will start via lab testing, and in parallel limited scale user engagement will start with focus groups, on the basis of early partial prototypes (**M9-M18**).
M24	2.0	The second prototype will be based on the feedback obtained after first deployment and will deliver a new release including complete functionality for most enCOMPASS components and interfaces, which will be evaluated at both usability and performance levels (yielding Release **R2**, **due at M24**). R2 will improve R1 with the features shown in Table 4. Both lab testing and user engagement will be increased exponentially
M36	3.0	This prototype is based on the results of second release and starts with activities for large scale demonstrators (at M25) leading to component and platform consolidation into the final version (**R3, due at M36**). R3 will provide the features shown in Table 5. Community feedback and user engagement will be at a superior scale and user behaviour evaluation and impact creation will be reported (**M25-M36**). Pilot activities will be exponentially intensified, to verify effectiveness of applications and fine-tune all results to ensure that enCOMPASS can generalize to other sustainability challenges and is ready for post-project application.

Table 1: enCOMPASS project initial planning

5.1 RELEASE NOTES

During the phases of the development process, each platform release will be accompanied by its release plan. The objective of a release plan is to check and validate the state of the project during the whole development life-cycle. Therefore, each **enCOMPASS** release is issued with a presentation document – the **Release notes** - with the goal to check and validate the implementation status of the release itself, and to update and add further details for the next development phase.

The **Release notes** document is scheduled to be released in conjunction with the correspondent software release delivery. In particular, Release notes have the following objectives:

- Reporting and evaluate the situation of the Development Process;
- Determining possible deviations from the initial Release Plan;
- Planning to implement necessary corrective actions;
- Refine and improve the Release Plan by refining its milestones.

The enCOMPASS release notes will conform to the template described in Table 2 below.

Release note identifier (name, date, version #)	
What's New	
System Requirements (third-party platforms / modules / etc with version numbers / dates)	
Features and changes (new features, defects corrected, caveats etc) Outstanding issues (unresolved defects, workarounds, installation issues etc)	
Installation guide (how to obtain and install)	
Known Issues	
Troubleshooting	
FAQ	
Other Resources and Links	

Table 2: template of the enCOMPASS release notes

5.2 CONTENT OF THE PLATFORM RELEASES

The following tables present the development phases, the release plan and the functionality included in the platform releases. The allocation of functionality is stable for release 1.0 and can be subject to variations for the later releases, due to the full specification of requirements and the feedback that will be collected after the trials and user evaluation of release 1.0.

Delivery Date	Release version	Features to be delivered
M18	1.0	Baseline definitions: energy consumption baseline data based on historical values (data sets described in D3.1 DATASETS WITH CONTEXT DATA AND ENERGY CONSUMPTION DATA).
		Smart meter and sensor data acquisition: processing and saving into the platform database of smart meter and sensor readings; connection with smart meter and sensor data.
		Initial building sensor data acquisition: database of sensor readings.
		Initial engagement methods and apps (initial configuration of the

		gamification engine, with basic stimuli actions and achievement).
		Initial behavioural change apps: initial mobile/web interfaces for login, logout, initial persuasive consumption visualization, inspection of recommendations, inspection of gamification data (leaderboard, points).
		Initial behavior modeler and personalized recommendation, based on the available user level input data.
		Initial persuasive games: proof of concepts of a board game for promoting energy awareness; proof of concept of a digital extension of the board game.
		Initial version of the gamification administrator web interface with basic view of the user's gamification activity in the consumers' web/mobile app.

Table 3: Release content of version 1.0 of the enCOMPASS platform

Delivery Date	Release Version	Features to be delivered
M24	2.0	Heterogeneous data fusion: smart meter, building, automation, gaming, social network integration.
		User modelling, classification and personalized recommendations.
		Modelling and initial impact assessment.
		Advanced behavioural change apps.

Table 4: Release content of version 2.0 of the enCOMPASS platform

Delivery Date	Release Version	Features to be delivered
M36	3.0	Large scale pilot testing.
		Engagement and behaviour change evaluation.
		Impact evaluation.

Table 5: Release content of version 3.0 of the enCOMPASS platform

6 TESTING STRATEGY

The Testing Strategy presents the testing procedures that will be performed in order to release the enCOMPASS platform. Testing is performed over the artefacts – components or services – that constitutes the enCOMPASS releases and platform as a whole. Each release is essentially composed by enCOMPASS applications – including automatic and human tasks, various components and enCOMPASS platform core services. The releases will be tested according to a double approach: by the component developer and by the platform integrator.

A typical scenario involves components providers that develop component which are further aggregated and orchestrated with the main platform. In this case, the Provider develops an artefact that are used by a Consumer (platform integrator) for the implementation of a specific module.

Once the components are tested these are delivered to the Integrator Partner (SETMOB) for packaging, in this sense the Integrator partner is the Consumer of the component. Also, the Integrator partner has the role of developer of the core services of the enCOMPASS platform.

Therefore, testing procedure has been organised in three main phases:

- **Artefact tests**: having the objective to find possible defects of each artefact of the enCOMPASS platform. This is the procedure applied in **Unit test** described in the following section.
- **Integration tests**: having the objective to tests the "coexistence" (the proper installation and operation) of the components developed by the enCOMPASS partners. The integration tests are in charge to the Integrator partner.
- **End to End functional tests**: having the objective to test an entire workflow, by testing a process from the very beginning all the way to the end.

All the testing phases are supported by the **Bitbucket** infrastructure and in particular by the section by the section Bugs, releases and feedbacks tracking system.

6.1 UNIT TESTING

Unit testing tests an artefact - component or service - before its official release, by means of functional tests performed to check the correctness of the component behaviour, following its functional specifications. Unit testing is not standardized within the enCOMPASS project, is applied to single components, interface and application.

Unit testing tests a component before its official release, by means of functional tests performed to check the correct component behaviour, following its functional specifications. Unit testing is not standardized within the enCOMPASS project, and is carried on under the responsibility of the component owner.

6.2 INTEGRATION TESTING

Integration testing tests the integration of a specific kind of artefact - component or service. Installation is checked in agreement with the installation requirements defined in the corresponding Release Note. Integration test is performed as a final test before enCOMPASS platform packaging. Integration testing is under the responsibility of the partner in charge of the Integration.

6.3 END TO END FUNCTIONAL TESTING

End to End Functional testing verifies the functionality of the integrated component and services deployed on enCOMPASS Platform. It has a dependency on the results of artefacts test. enCOMPASS modules, decomposed in use cases, are exploited using the User Interface as test sessions. Functional testing is performed mainly by the Integrator partner, supported by all the other partners involved.

Any testing session calls one of the entry points of the prototype and checks desired response.

Component or service testing can be in charge of component and service owner and/or Platform integrator. Initial test is performed by component or service owner for smooth testing. The completed functional testing is performed by platform integrators supported by component or service owner for all the components and services involved.

The relationship between use cases and test sessions could not be one-to-one, because a complex use case could be made by several distinct ways to interact with the enCOMPASS Platform, and each one must be subject to a specific testing session.

6.4 TESTING TOOLS

Specific software tools will be used for automating the testing process of the enCOMPASS platform and its components. The purpose is to control the execution of all the necessary tests and to confront the actual outcomes with predicted outcomes. Automating tools can automate significant parts of the repetitive tasks foreseen by the testing process or add additional testing that would be difficult to perform manually.

6.4.1 GUI testing

User interface (UI) testing is the process used to test if the application graphic interface is functioning correctly. UI testing can be performed manually by a human tester, or it can be performed automatically with the use of a software program as **Selenium**.

Selenium allows scaling for a large number of tests, or for tests that must run in multiple environments. It also allows to run different tests simultaneously on different remote machines.

6.4.2 Web service testing

In the architecture of the enCOMPASS platform, the main role of the web services is achieving the component integration. When the number of the web services becomes significant, a key issue is ensuring their functional quality while avoiding to introduce opportunities for error or failure. Such automated tools for web service testing are:

- **SoapUI** [SoapUI] represents a functional testing solution for web services. SoapUI allows creating and running automated functional, regression, compliance, and load tests. In a single test environment, it provides complete test coverage and supports all the standard protocols and technologies.
- **JMeter** [JMeter] is an automated tool to test performance both on static and dynamic resources. It w be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

6.4.3 Component testing

JUnit [Junit] is an open source unit testing framework for the Java programming language. This framework will be integrated in the development of the business logic within the platform components. It allows the

developer to incrementally build test suites to measure progress and detect unintended side effects. Tests can be run continuously. Results are provided immediately.

6.4.4 Performance testing

Webserver Stress Tool [PAESSLER] is a HTTP-client/server test application designed to detect critical performance issues at the web sites level thus ensuring optimal experience for the users. By simulating the HTTP requests generated by a lot of simultaneous users, this tool will test the consumer and utility portal performance under normal and excessive loads.

7 CONCLUSIONS AND FUTURE WORK

This deliverable has presented the current status of the work regarding the ground for an effective collaboration environment for developing, testing and releasing the enCOMPASS software platform, which is the output primarily of WP6 platform Implementation and Integration. It has requested coordination and integration of input from all the active work packages.

The ongoing actions consider:

- Finalizing and delivering the detailed architecture design of the enCOMPASS platform describing all platform modules (e.g., components, services, and applications), communication protocols, and underlying information and data models.
- Implementing the Continuous Integration software development environment as a must-have condition for collaboration projects in a loosely coupling environment. Continuous integration tools allow testing, compiling, installing and auditing by an automated build, allowing teams to detect problems as early as possible.
- Finalizing and fine-tuning for the enCOMPASS data base schema and instance.
- Initiating the development for the Smart Meter data acquisition component. This component will ensure the population of the database with test data from the demo use cases at TWUL and SES.
- Initiating the development of the Web GUI for allowing end-users to login and register their data not provided by sensors. (e.g., data for identifying houses, users, billing prices etc.)
- Initiating the development of the Web services for integrating gamification mechanism into other applications, produced either by enCOMPASS or by third parties.
- Initiating the development of the Web-Service interface for importing data conforming to the enCOMPASS gamification model (e.g., user action logs in business applications, users' achievements in mobile digital games, etc.).

Future work will address the development and testing of the components included in the first prototype (R1) of the enCOMPASS platform. The modules of the first prototype will be continuously compiled, installed, configured, and deployed. In the same time, feedback on the platform's usability, coming from the validation case studies will be considered to permanently improve the prototype's capabilities.

8 REFERENCES

- [Ambler04] Scott W. Ambler, The object primer: Agile model-driven development with UML 2.0, 2004, Cambridge University Press
- [Bitbucket] Free source code hosting for Git and Mercurial. [Available online at https://bitbucket.org]
- [Bugzilla]. The Web-based general-purpose bug tracker and testing tool. [Available online at http://www.bugzilla.org]
- [BCW12] Marco Brambilla, Jordi Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice (Synthesis Lectures on Software Engineering) Paperback – September 26, 2012
- [BF14b] Marco Brambilla, Piero Fraternali: Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience. Sci. Comput. Program. 89: 71-87 (2014)
- [CBB03] Piero Fraternali, Marco Brambilla, Aldo Bongio, Sara Comai Stefano Ceri Designing Data-Intensive Web Applications (Dec 1, 2003)
- [Carrol95] Carroll, J.M. (1995). Introduction: The Scenario Perspective on System Development. In J.M. Carroll (ed.) Scenario-Based Design: Envisioning Work and Technology in System Development. New York: John Wiley & Sons.
- [Cockburn01] Cockburn, A. (2001). Writing effective use cases. Addison Wesley, 2001. ISBN 0-201-70225-8.
- [cPrime] cPrime Inc.. Introduction to SCRUM for Project Managers. [Available online at http://www.slideshare.net/montemontoya/agile-scrum-essentials-for-project-management]
- [Chen76] Peter P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems (TODS) , Volume 1, pages 9-36
- [FB2014] Marco Brambilla, Piero Fraternali, Interaction Flow Modelling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML (The MK/OMG Press) Paperback – December 14, 2014 ISBN-13: 978-0128001080 ISBN-10: 0128001089
- [HIB] Hibernate ORM. Idiomatic persistence for Java and relational databases. [Available online at http://hibernate.org/orm/]
- [IBM-SOA] [Available online at http://www.ibm.com/developerworks/library/ws-soa-design1]
- [IBM-SOMA] Available online at https://www.ibm.com/developerworks/community/blogs/AliArsanjani/entry/soma_a_method_for_developing?lang=en
- [ISO99] ISO 13407. Human-centred design processes for interactive systems. ISO, 1999.
- [Jenkins]. The leading open-source continuous integration server. [Available online at http://jenkins-ci.org]
- [JMeter] Apache JMeter. A 100% pure Java application for testing functional behaviour. [Available online at http://jmeter.apache.org]
- [JUnit] About Junit. [Available online at http://junit.org]
- [Maven]. The software project management and comprehension tool. [Available online at http://maven.apache.org]
- [Nexus]. Repository manager for artefacts. [Available online at http://www.sonatype.org/nexus]
- [SCRUM METHODOLOGY] Scrum Methodology [Available online at http://scrummethodology.com)]
- [OMG-SOAML] [Available online at http://www.omg.org/spec/SoaML]
- [OMG-UML] [Available online at http://www.omg.org/spec/UML/]
- [OMG-IFML] [Available online at http://www.omg.org/spec/IFML/]

- [PAESSLER] Web Server Stress Tool. Performance, Load and Stress-Test for Web Servers.[Available online at http://www.paessler.com/webstress]
- [Siegel05] Siegel, J. (2005). Introduction to OMG UML. OMG. [Available online at http://www.omg.org/gettingstarted/what_is_uml.htm]
- [Spring] Spring Framework. Core support for dependency injection, transaction management, web applications, data access, messaging, testing and more. [Available online at http://projects.spring.io/spring-framework/]
- [Selenium]. Web brower automation. [Available online at http://www.seleniumhq.org]
- [SoapUI] The Swiss-Army knife for testing. [Available online at http://www.soapui.org]
- [Sonar]. Continuous code quality management. [Available online at http://sonarsource.com]
- [ThoughtWorks] Continuous Integration [Available online at http://www.thoughtworks.com/continuous-integration]