Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Requirements Engineering Framework, Language and Tools for Service Platforms

Abstract

This document presents the methodologies, languages, and tools proposed in INDENICA for the holistic elicitation of requirements. The document is split into four parts: Part 1 is the Requirement Engineering process derived from Product Line Engineering and transferred to Platform as a Service, Part 2 is IRENE, the goal-based solution for stating the actual requirements of the service platform, Part 3 is about user-centric Requirement Engineering, and Part 4 deals with Return on Invest calculation.

Document ID:	INDENICA – D1.2.1
Deliverable Number:	D1.2.1
Work Package:	1
Type:	Deliverable
Dissemination Level:	Public
Status:	Final
Version:	1.4
Date:	2011-09-30
Contributing Partners:	SIE and PDM

Version History

0.1	01 July 2011	Initial version
1.0	31 August 2011	Preliminary content
1.1	05 September 2011	IRET added
1.2	10 September 2011	Almost complete version
1.3	17 September 2011	Version for internal review
1.4	30 September	Final Version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

List of Acronyms

AE	Application Engineering
DE	Domain Engineering
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
IRENE	Indenica Requirements ElicitatioN mEthod
IRET	IREne Toolset
LTL	Linear Temporal Logic
PaaS	Platform as a Service
PLE	Product Line Engineering
RE	Requirements Engineering
PO	Product Owner
ROI	Return On Invest

1 Introduction

This document presents the methodologies, languages, and tools proposed in INDENICA for the holistic elicitation of requirements. The document is split into four parts: Part 1 is the RE process derived from PLE and transferred to PaaS, Part 2 is IRENE, the goal-based solution for stating the actual requirements of the service platform, Part 3 is about user-centric RE, and Part 4 deals with RoI calculation.

As for the goal-oriented solution, the document proposes IRENE (Indenica Requirements ElicitatioN mEthod). This proposal borrows from KAOS, which is a well-known requirements elicitation framework, and also from some previous work done at Politecnico di Milano, to provide the user with a “complete” solution to elicit the “usual” functional and non-functional requirements, but also to state the foreseen adaptation capabilities and also the requirements in terms of the variability that should be embedded in the system-to-be. Moreover, requirements can be classified as either crisp, that is, they are either satisfied or not satisfied, and fuzzy, to embed flexibility in the system and be able to reason in terms of different degrees of satisfaction. The idea is to offer a comprehensive and homogenous solution to let users specify the requirements of their INDENICA platforms, even if the proposal can cover a wider spectrum.

IRENE offers a set of graphical symbols to let the user state the structure of the requirements, and also some textual annotations to refine and better specify the concepts. As for this last aspect, IRENE can be used in two different ways. Annotations can be added in the form of natural language, to ease the user in his/her work, but also to allow for an incremental specification of requirements, but they can also be stated using formal notations. The current version of IRENE is prescriptive as far as requirements and adaptation are concerns, while it is still open as for variability.

Even if IRENE could easily be used to specify single applications, the focus on service platforms imposes a methodological shift. We think that a platform could be specified as if it were a “conventional” single solution, whose aim is to provide services to others, which in turn may create different applications. But, it could also come from the identification of some “reference” applications, which are then used to generalize the concepts and define a single and coherent specification. In the first version of IRENE, this blending process is mainly in the head of the analyst, but we plan to extend it and provide suitable solutions in the next (and final) version.

A prototype tool called IRET (IREne Toolset) supports all these aspects. IRET is implemented as an Eclipse plugin and fully supports IRENE to allow users easily define complete and coherent requirements models.

For User Centric Requirements Engineering we focused on the definition and on methods how to put the user into the centre of the elicitation and prioritization process. We propose a new visualization for the prioritized requirements to get better overview for the overall priority of the requirements and for the decision about platform or application requirement. Also SCRUM as agile method is analysed how it focuses on the user and how Product Line Engineering (PLE) and platform development could be integrated into agile development using SCRUM.

Return on Invest (ROI) calculations are basic for decision support, whether an organization will spend money on a new system or not. The ROI Estimation method for Service Platforms is derived from the methodology developed for Product Lines. It is extended by a set of additional parameters. It allows estimating the base figures and bringing them into a structured context, represented by a set of formula. The result is an estimated Return on Investment for introducing a platform approach. The method allows to clearly identify assumptions and thus helps to understand the influencing factors, identify risk and thus support the decision process when introducing a service platform.

The rest of the document is organized as follows. Section 2 provides a brief introduction to requirements engineering for product lines in the context of INDENICA and frames the document. Section 3 introduces IRENE, along with its tool support called IRET. Section 4 introduces user centric requirements engineering. Section 5 presents the methodology for estimating the return on investment, and Section 6 concludes the document.

2 Requirements Engineering for Platforms as a Service

As described in [SotA], the main characteristics for Product Line Engineering (PLE) consist in:

- The existence of two different development processes:
  - o Domain Engineering (DE):
    The process of software product line engineering, in which the commonality and the variability of the product line are defined and realized.
  - o Application Engineering (AE):
    The process of software product line engineering, in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability
- Variability as a core concept for PLE
- A platform for the product line
- A reference architecture

As one main principle of PLE is building a platform we propose to transfer requirements engineering for PLE to the virtual service platform of the Indenica project. In the Indenica context, the applications of the product line would be the different applications using the services provided by the platform.

Figure 2.1 shows Requirements Development within PLE process.

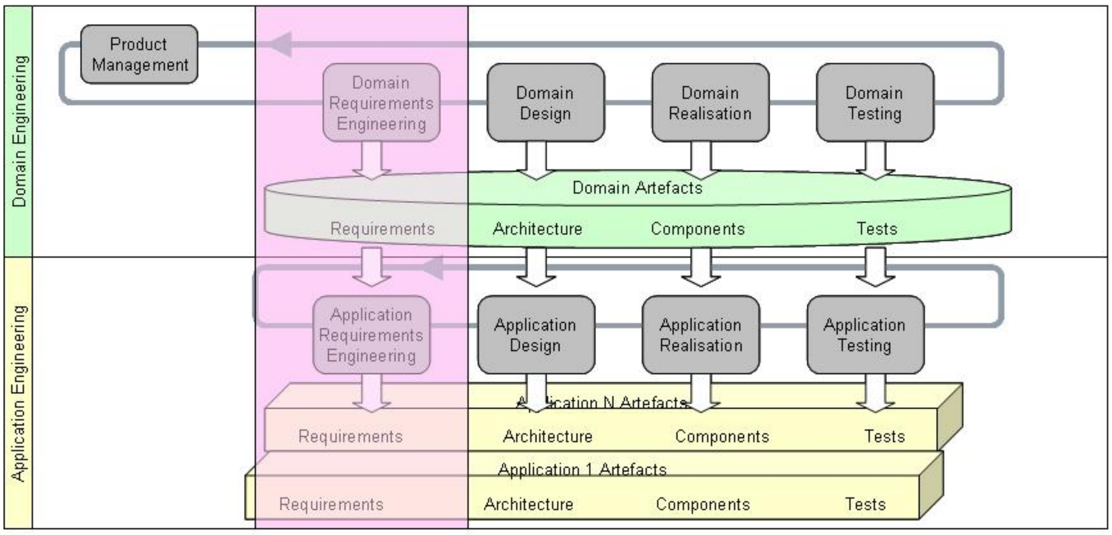


Figure 2.1 Domain and Application Requirements Development

For all following considerations regarding “Requirements Development in the context of PLE” we define the workflow in Figure 2.2 as a process basis:

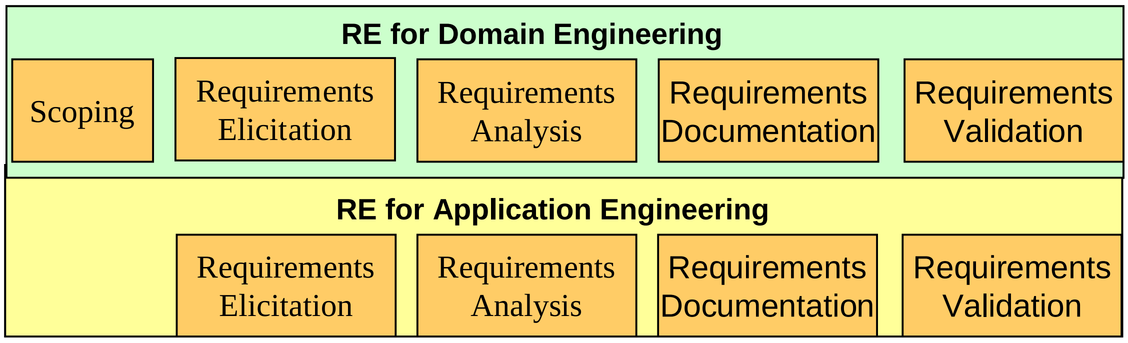


Figure 2.2 Requirements Development as process basis for PLE

In the following sections the RE process for product lines is described in more detail, with challenges for RE arising from product line context. This is the basis for further research.

2.1 *RE for Domain Engineering*

Domain Engineering is a specific discipline necessary when developing product lines. In the Indenica context this discipline will be used to design the virtual service platform

The domain engineering process deals with the development of all parts of the product line which are common to all applications of the product line or a specific number of diverse applications of the product line. This has effects on different steps of the requirements engineering process.

2.1.1 Requirements Management

The main purpose of requirements management in the context of Domain Engineering is to ensure the consistence of requirements common to all related applications.

Major tasks are consequently:

- Tracing requirements back to their origin
- Managing cross-references between requirements
- Tracing requirements forward to their implementations
- Managing requirements changes

2.1.2 Requirements Development

The main purpose of requirements development in the context of Domain Engineering is to evolve the requirements common to all related applications or a well-defined subset of those applications and to document them in a structured manner.

All activities (already well-known for requirements development within product development) have to be fulfilled on a higher degree of complexity.

Additionally to those tasks (known for general requirements development), Domain Engineering requires an additional process step.

All these specific characteristics are described in the following chapter.

Scoping:

First of all it is crucial to define the scope for domain engineering (the platform system or core asset base and its boundaries).

Based on the results of the analysis for goals and strategies of the product portfolio (roadmap and strategy), the product line manager has to determine the scope for the platform.

Output of this process step is a "Product Line Strategy", which is the base for the following Domain and Application Engineering Process.

Elicitation:

The main challenges concerning requirements elicitation within domain engineering are [Pohl et al. 2005]:

- Identification of requirement sources for the whole product line
  First challenge in eliciting requirements for the whole product line is to identify the range of requirements sources. A promising approach for identification is to consider all already existing applications and their requirements to the product line, as in most cases product lines are developed based on already existing applications.
  As a product line may cover many diverse applications, this range of requirements sources possibly appears very heterogeneous and even contradictory.
  This leads to the next challenge:
- Identification of common requirements (commonality analysis)
  The central benefit of PLE is the commonality of artefacts across diverse applications. For requirements engineering this implies the existence of common requirements for several applications.
  - A proven method for the evaluation of common requirements is the "application - requirements matrix". In this matrix it is listed which application is effected by which requirement. Requirements which appear for all applications may be identified as common requirements.
  - Another method is the priority-based analysis:
    Different stakeholders are asked to prioritize a given set of requirements. A specific algorithm calculating the relevance of each requirement for the whole product line may give indication of the availability of common requirements.
- Identification of variable requirements (variability analysis)
  The complementary analysis to commonality analysis is the variability analysis. It is performed using the same methods as commonality analysis:
  - application - requirements matrix
  - priority based analysis
    Requirements which are assigned only to a few applications or are high-rated only by a few stakeholders may be identified as variable requirements.
- Monitoring of requirements along their lifecycle
  Requirements may change over their lifecycle from common to variable require-

ments. So it is important to monitor the requirements, if the classification of the requirements is still valid.

Analysis:

The main challenges concerning requirements analysis within domain engineering are [Pohl et al. 2005]:

- <u>Consistency Check</u>
  see [Lauenroth/Pohl]
- <u>Defining Requirements Variability</u>
  Based on the results of the commonality and variability analysis, it is essential to develop a variability model which illustrates:
  - variants
  - variation points
  - variability dependencies

Documentation:

The main challenges concerning requirements documentation within domain engineering are:

- <u>Documenting variability in requirements artefacts</u>
  As well as in general requirements engineering, there are different ways of documenting requirements. In domain engineering the additional challenge lies in the documentation of variability:
  - Natural-language documentation
    It is helpful to use graphical elements
  - Documentation by graphical models
    - Describing requirements variability in a feature tree
    - Describing requirements variability in a Use Case model
    - Describing requirements variability in other models
- <u>Tracing between artefacts and model</u>
  For all options it is essential to define and maintain the traceability from variability model to the chosen artefact.

Validation:

The main challenges concerning requirements validation within domain engineering are:

- <u>High quality for common requirements</u>
  As common requirements have an immense impact on several or even all applications of a product line, the requirements engineer in domain engineering has to focus especially on the quality of the requirements.
  The quality aspects for general requirements engineering are valid to a special degree.
  Due to the complexity in domain engineering, it is an even bigger challenge to comply quality attributes as "unambiguous" or "consistent".

- Validation of variants
  Depending on the relevance of requirements for specific variants, the specific validation attributes of the requirements have to be set accordingly.
  Thus two effects can be noticed:
  - All relevant scenarios are covered by validation -> all variants and their combination can be validated
  - Unrealistic scenarios are not considered by validation -> no need for useless excessive validation efforts

2.2 RE for Application Engineering

The main characteristic for requirements engineering within application engineering consists in the presence of already existing requirements artefacts arisen from domain engineering. In the INDENICA context this will be the virtual service platform. The application build by application engineering will be the applications using the services from the platform.

These results serve as essential input for all RE activities within application engineering, as the main target of product line engineering is to reuse as many artefacts built in domain engineering as possible for application engineering.

Thus for requirements engineering that implies the reuse of existing RE artefacts from domain engineering as [Pohl et al. 2005]:

- Common requirements
- Variable requirements

INDENICA Work Package 2 focus on variability modelling in the specific Indenica service oriented context.

2.2.1 Requirements Management

Requirements Management for Application Engineering has to ensure the consistence of all requirements of each specific application to

- other requirements of the same application
- the requirements of the core asset base
- the implementations within the applications
- the changes along the product lifecycle

2.2.2 Requirements Development

Elicitation:

The main challenges concerning requirements elicitation within application engineering are [Pohl 2005]:

- Communication of requirements artefacts from domain engineering to the stakeholders
  Different from elicitation for single product development, in application engi-

neering it is necessary to inform all relevant stakeholders of the applications about the variety of already existing common requirements and variable requirements assigned to specific variants and the related variability model.
Based on these prerequisites the next activity has to be executed:

- Establishing a set of variants appropriate for specific application
  Using now this input from domain engineering, it is necessary to select - beside the mandatory common requirements - appropriate variable requirements from different variants.
  After this evaluation a certain set of requirements - common requirements and variable requirements arisen from different variants - is defined which already fulfil a certain percentage of the original requirements of the stakeholders.
  The remaining requirements not covered by the selected domain requirements are considered in the next step:
- Establish the delta between domain and application requirement artefacts:
  Usually not all requirements of the application can be satisfied by domain requirements. These application specific requirements have to be elicited additionally to the domain requirements and are subject of the subsequent analysis activities.

Analysis:

The main challenges concerning requirements analysis within application engineering are [Pohl 2005]:

Additionally to the analysis activities like "Consistency checking", as described in the chapter of domain engineering, it is essential to regard the relationships between domain and application engineering activities and artefacts. Especially the analysis of deltas between the domain variability model and the application variability model is crucial for the further engagement in requirements analysis:

- Impact analysis for deltas between application requirements to existing variability model w.r.t. existing variation points:
  - If for an already existing variation point a specific option for this application is missing, it might be necessary to add a new variant to an existing variation point.
  - If an existing variation point does not cover the correct variants, it might be necessary to modify these existing variants
- Impact analysis for deltas between application requirements to existing variability model w.r.t. common requirements:
  A common requirement might convert from common to variable, if a new aspect - not yet considered in variability model - has been introduced. So a new variation point has to be added.
- Decision about implementation of deltas
  For each identified delta, it has to be decided, if this new part of the variability model will be developed. Here, structuring and prioritizing activities as described in the part "Domain Engineering" might be helpful.

Documentation:

The main challenges concerning requirements documentation within application engineering are [Pohl et al. 2005]:

- Documenting all requirements used from domain engineering:
  Either the common requirements mandatory for all applications of this product line either the variable requirements of the domain variability model used for this application have to be described here.
- Delta between domain and application requirements
  All requirements which have not been derived from domain variability model (requirements which are new or modified)
- Application variability model:
  Also the variability model of this application with its variants has to be described.

3 IRENE

This section introduces the first version of IRENE, the Indenica Requirements ElicitatioN mEthod, which we propose for the definition/elicitation of the requirements of innovative service platforms. IRENE is centred on an extended goal-based solution for requirements elicitation.

IRENE uses the KAOS model [van Lamsweerde 2009] as a starting point to represent requirements. This choice is due to the flexibility of goal-oriented methodology, and also to their ability to show the alignment of the system-to-be with stated objectives through the refinement relationship among goals. Furthermore, the representation of the goals' operations eases the derivation of the functional representation and the application, but also of the monitoring directives that we want to deploy at runtime. IRENE extends KAOS is some different directions:

- It adds the notation of *adaptation goal* to emphasize the importance of adaptation and evolution in modern software systems (service platforms) from the very beginning of the design process;
- It introduces the notion of *crisp* and *fuzzy* goals to distinguish between requirements whose satisfaction is either fully true or false, and requirements that can also be partially satisfied. Fuzzy goals allow us to quantify the degree x (x is between 0 and 1) to which a goal is satisfied/violated, and provide the flexibility necessary in systems where some goals cannot be clearly measured (soft goals), properties are not fully known, their complete specification –with all possible alternatives– would be error-prone, and small/transient violations must be tolerated;
- It complements the different notation elements with further information about variability. In this first version, IRENE only supports annotations that can be associated with the different modelling elements to let the analysts concentrate on the actual variability that they want to embed in the running solution;
- It introduces the notion of *family* of related specification to better support the idea of family of applications that in the end must be supported by the same platform. This concept is only introduced in this first version of IRENE, but it will be fully developed in the next version.

Adaptation goals can be specified at different degrees. *Foreseen adaptations* are fully specified, since they represent the corrective actions that must be performed for known problems. *Partial adaptations* are adopted to devise undesired situations for which a corrective action cannot be identified at design time, but can be only identified dynamically, depending on the current execution context. Finally, *default adaptations* are adopted to solve cases in which foreseen adaptations are not effective enough, or the way the application designed is not satisfactory to achieve the global satisfaction of stated goals.

3.1 Requirements elicitation

IRENE supports the elicitation of requirements for service platforms in different ways. It is prescriptive in the set of graphical elements offered to the user to shape the requirements specification, but then it becomes quite flexible and liberal in the way these elements can be used.

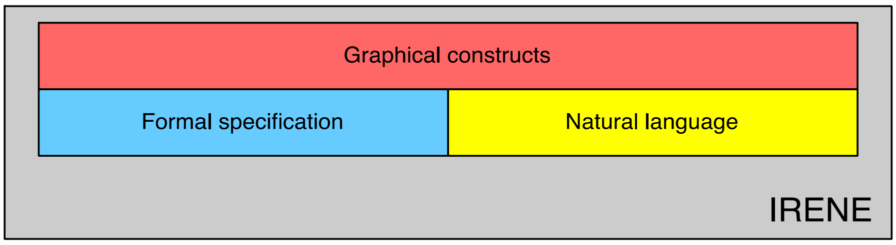


Figure 3.1: The two levels of IRENE.

Figure 3.1, explains the two levels of “rigour” offered by IRENE. Users are free to shape their requirements and provide details in natural language, but they can also add formal definitions of the different concepts. This two-step process helps users work incrementally, but the latter also enables validation capabilities and automatic transformations of specified artefacts. Any possible intertwining of the two levels is admitted, but the two extremes (natural language only or complete formalization) should be preferable. It is true however that the analyst may decide to only provide informal descriptions of easy/well-known parties and concentrate on ---formalize--- the newest, most complex, and most obscure ones.

IRENE is a conventional KAOS-based requirements elicitation notation if the user only considers “conventional” requirements. It becomes a bit more sophisticated if the user distinguishes between crisp and fuzzy requirements (see Section 3.3); it is even more complete when adaptation is taken into account (Section 3.4). Moreover, since the INDENICA solution provides explicit variability modelling, requirements must only capture the “variability requirements”, this means that the model must offer the capability of identifying the needs as for variability is concerned, but not the actual variability models, which will come after the engineering phase.

Each requirements specification can be organized in different ways. One single model is usually enough for small-to-medium systems, while different models are mandatory for large systems. Given the hierarchical nature of IRENE models, the user can easily identify suitable slices and deal with them independently. Usually, the high-level goals define the first view on the system; then the other goals can be developed in specific models. Again, IRENE does not prescribe any particular template for organizing requirements, but the users are free to identify the set of models that fit their needs.

Since IRENE has been conceived to ease the identification of the requirements of service platforms (both platforms that offer services and platforms as a service), the user can follow different processes to provide a complete and coherent specification of the requirements of the platform-to-be. In some cases, the elicitation can easily be organized around a single model, or a hierarchy of models as described above. It is also true that the requirements for a complete platform may come after modelling (some of) the applications it should support. This is why IRENE also envisions the idea that a single specification in reality originates from some different/independent models. The actual fusion of these different models could not be easy since different models (applications) may present conflicting or incomplete requirements. Currently, IRENE does not provide any special purpose solution to ease the assembly, but we plan to provide better support in the next version.

3.2 KAOS

A KAOS model comprises four sub-models: goal model, object model, agent responsibility model, and operation model, which are related through inter-model consistency rules. They ease the reasoning about the different views of the system and the generation of a direct mapping onto the underlying implementation.

A goal is an objective (of some stakeholders) for the system, which includes both the software and its environment. The goal model defines the main objectives the application should meet. It defines a hierarchy of goals that relates the high-level goals to low-level system requirements. Goals can be refined into conjoined sub-goals (AND-refinement) or into alternative combinations of sub-goals (OR-refinement). The satisfaction of the parent goal depends on the achievement of all (for AND-refinement) or at least one (for OR-refinement) of its underlying sub-goals. Goals decomposition can be accomplished by means of formal rules [15]. The refinement of a goal terminates when it can be "operationalized" or, in other words, it can be decomposed into a set of operations.

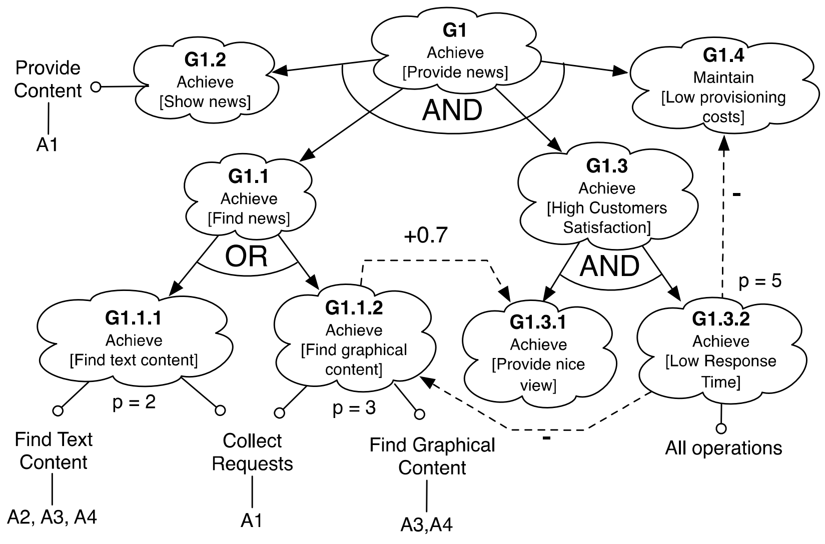


Figure 3.2: Example goal model.

Goals are formally expressed in terms of LTL (Linear Temporal Logic [Pnueli 77]) expressions[1]. Each goal can follow a particular pattern depending on its temporal behaviour: achieve/cease goals are specified through sometimes in the future/past operators, while maintain/avoid goals are specified through always in the fu-

[1] They can be specified through operators like sometimes in the future (◇), sometimes in the past (◆), always in the future (□), always in the past (■), always in the future unless (W), always in the past back to (B), always in the future until (U), and always in the past since (S).

ture/past operators. While KAOS prescribes users to state goals in LTL, we think that this would be too much for IRENE (at least in the first version), and thus we leave the user the freedom to either specify goal formally or simply identify their main elements through natural language. Needless to say, the subsequent “use” of these goals will be very different in the two cases.

Figure 3.2 shows the KAOS goal model of a domain-specific platform in charge of providing services about news. The general objective of the platform is to provide news to its customers (G1), which is AND-refined into sub-goals: Achieve[Find news] (G1.1), Achieve[Show news] (G1.2), Achieve[High customer satisfaction] (G1.3), and Maintain[Low provisioning costs] (G1.4), expressed in terms of the number of servers used to provide the service. News can be provided both in text and graphical mode (see OR-refinement of goal G1.1 into goals G1.1.1 and G1.1.2). Text mode consumes less bandwidth and performs better in case of many requests. Customer satisfaction is increased by providing news in a nice way and within short response times (see AND-refinement of goal G1.3 into goals G1.3.1 and G1.3.2).

Goals G1.3 and G1.4, together with their refining goals, represent non-functional requirements. Goal G1.3.1 is a *soft* goal, since there is not a clear-cut criterion to assess whether news have been provided in a nice way. While goal G1.3.2 and G1.4 can be satisfied in different ways depending on the distance between the actual response time and the maximum tolerated delays (for goal G1.3.2), or the difference between the number of used servers and the number of servers available.

Goals are associated with a priority depending on their criticality. For example, goal G1.1.1 has lower priority (p = 2) than goal G1.1.2 (p = 3), since providing news in graphical mode is more important than providing news only in text mode. Goals can also contribute (positively/negatively) to the satisfaction of other goals. This relation is represented in the goal model through contribution links ---dashed lines in Figure 1--- and an indication of the contribution (with x between -1 and 1). For example, despite the graphical mode is slower, it positively contributes to the customer satisfaction (contribution link between goals G1.1.2 and G1.3.1). Short response times may require the adoption of the text mode to provide the news (see conflict between goal G1.3.2 and goal G1.1.2) or may increase the provisioning costs since they may require a higher number of servers.

Even if IRENE is less prescriptive than KAOS, to provide the reader with a „complete and consistent” example, Table 3.1 presents the formal definitions of the goals of Figure 3.2. For example, goal G1.1.2 asserts that after a request is received provided news must be about supplied keywords and date, and must come with images.

Goal	**Description**
	Formal Definition
G1.1.1	After a request is received (event *ReceiveRequest*), provided news (role *news* of association *NewsCollection*) must be about supplied keyword and date.
	$nc : NewsCollection,\ ReceiveRequest(keyword,\ date,\ user,\ pwd) \wedge \neg(\exists\, r \in nc.request) \wedge (keyword \neq \text{“”} \vee date \neq null) \Longrightarrow \Diamond_{t<x}(\exists\, n \in nc.news \mid (n.keyword = keyword \vee n.date = date) \wedge n.text \neq null)$
G1.1.2	After a request is received (event *ReceiveRequest*), provided news (role *news* of association *NewsCollection*) must be about supplied keyword and date, and must come with images.
	$nc : NewsCollection,\ ReceiveRequest(keyword,\ date,\ user,\ pwd) \wedge \neg(\exists\, r \in nc.request) \wedge (keyword \neq \text{“”} \vee date \neq null) \Longrightarrow \Diamond_{t<x}(\exists\, n \in nc.news \mid (n.keyword = keyword \vee n.date = date) \wedge n.text \neq null \wedge (\exists\, i \in n.images))$
G1.2	After news are collected, they must be provided to the user (event *ShowNews*) within y time instants
	$nc : NewsCollection,\ n : News\ (\exists\, n \in nc.news) \Longrightarrow \Diamond_{t<y} ShowNews(news) \wedge nc.news = news$
G1.3.2	The end to end response time of the News Provider must be less than RT_{MAX} time instants
	$nc : NewsCollection,\ r : nc.request,\ ReceiveRequest(keyword,\ date,\ user,\ pwd) \wedge \neg(\exists\, r \in nc.request) \wedge (keyword \neq \text{“”} \vee date \neq null) \Longrightarrow \Diamond_{t<RT_{MAX}}(ShowNews(news) \wedge nc.news = news)$
G1.4.1	The number of servers must be always less than N_{MAX}
	$servers : int,\ servers \leq N_{MAX}$

Table 3.1: Formalization of the goals of Figure 1

The vocabulary used in the goal model introduces the main elements of the platform. These elements are fully specified in the object model. Objects can be: entities, associations, events, or agents. Each object is characterized by the following attributes: name, identifier, type (entity, event, agent, association), domInv, which lists known domain properties about the object as invariants holding in any object state; and init, which states the initial value of its attributes and associations. The object model for the new platform is shown in Figure 3.3.

Entities are objects providing informative content. The previous goal definitions introduce entities: News, Request, and Customer. Each entity is characterized by a unique identifier (id). News is qualified by text content (attribute text), and a set of images (attribute images). Request is described by a topic (keyword) and a date (date). Customer represents the user issuing a request, and s/he is characterized by a name (user) and a password (pwd).

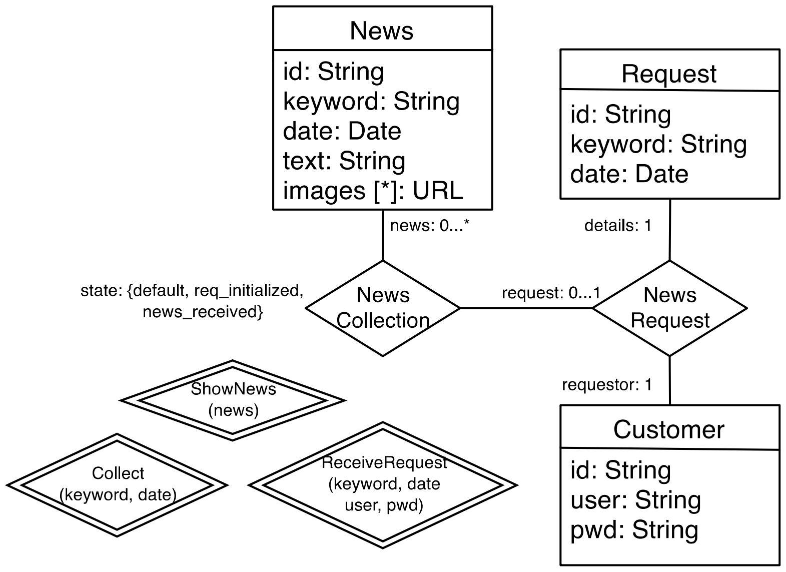


Figure 3.3: Object model of the example system

Associations are conceptual links between objects, where each linked object plays a specific role in the association. A multiplicity is given for each role that specifies the minimum and maximum number of object instances that can be associated. They also have cardinality, which is the number of objects linked by it[2]. The associations devised for our example are NewsRequest and NewsCollection. The former links a customer to the request he/she has issued; the latter links a request to a set of matching news.

Both entities and associations can evolve through a set of states, depending on the values assumed by their attributes. In the example, all entities/associations have only one state, with the exception of association NewsCollection that has a *default* state, in which its news are empty and its request has not been initialized yet, state *req_initialized*, when the customer has already issued a request (i.e, role *request* is initialized, and state *news_received* when news are collected (i.e., role *news* is not empty).

Events are instantaneous objects, corresponding to something that may happen during the execution the services provided by the platform. As for events, the example requires: ReceiveRequest, that is, a request is issued by the user, Collect, that is, news can start to be collected, and ShowNews, that is, news are sent to the user. These events and their corresponding attributes are reported in Figure 3.3.

Agents can be software components, external devices, and humans in the environment, responsible for the satisfaction of some goals. Agents[3] can monitor and control

[2] Associations can be reflexive, when the same object appears in the association at multiple positions under different roles.

[3] An agent monitors an object if it can get the values of the object's attributes. An agent controls an object if its instances can modify the object's attributes. Finally, an agent controls an association if can create/delete the association instances.

entire objects or only some of their attributes, as shown in Figure 3.4. For example, agent A4 can monitor association NewsCollection and can modify its news (including images).

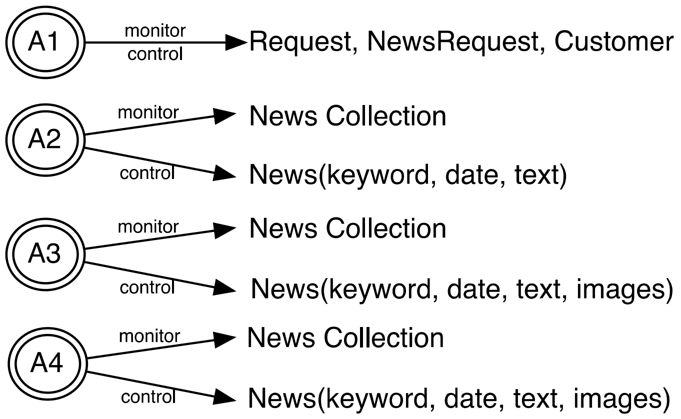

A1
monitor
control
Request, NewsRequest, Customer
A2
monitor
News Collection
control
News(keyword, date, text)
A3
monitor
News Collection
control
News(keyword, date, text, images)
A4
monitor
News Collection
control
News(keyword, date, text, images)

Figure 3.4: Agents for the example system.

Different agents can be responsible for the realization of a goal, depending on the objects they can monitor/control. This way, the agent responsibility model assigns goals to agents, responsible for their achievement (see Figure 3.2). As for agents, agent A1 is the user issuing the requests and to whom news must be shown. Agents A3 and A4 can find news in both text and graphical mode, while agent A2 can only find news in text mode. In our platform, an agent will be a service provided by the platform itself. For example we foresee a service *Retailer*, which offers products (to fulfill goal *Achieve[ProductFound]*), a service *Payment*, responsible for electronic payments (goal *Achieve[CCPayment]*) and one or more mobility services, allowing users to book a delivery method (goal *Achieve[Delivery]*).

Leaf goals are also associated with the operations necessary for their achievement (operation model). An operation is an input-output relationship over a set of objects. Operations are specified by their effects on the domain: domain pre- and post-conditions (*DomPre* and *DomPost*). A domain pre-condition characterizes the state before applying the operation; a domain post-condition defines a relationship between the state before and after applying the operation. Operations are also specified through required pre-conditions (*ReqPre*), triggering pre-conditions (*TrigPre*) and required post-conditions (*ReqPost*). Required pre-conditions define those states in which the operation can be applied. Triggering conditions define those states in which the operation must be immediately applied, provided the domain precondition is true. Required post-conditions define additional conditions the application of the operation must satisfy.

The news provider must support the following operations: *Collect Requests*, *Find Text Content*, *Find Graphical Content*, and *Provide Content*. The definition of these operations is provided in Table 3.2. The definition of operation Find Text Content is similar to operation *Find Graphical Content*, except for the required post-condition.

Operation	Definition
Collect Requests	**Input:** $nc: NewsCollection$ **Output:** $nc: NewsCollection$ **DomPre:** $nc.state = default$ **DomPost:** $nc.state = req\_initialized$ **ReqPre:** $\neg(\exists\, r \in nc.request) \wedge$ $(keyword \neq$ "" $\vee\, date \neq null)$ **TrigPre:** $ReceiveRequest(keyword, date, user, pwd)$ **ReqPost:** $(\exists\, r \in nc.request \mid r.details.keyword = keyword \wedge$ $r.details.date = date \wedge$ $r.cust.user = user \wedge r.cust.pwd = pwd$
Find Graphical Content	**Input:** $nc: NewsCollection,\ r: nc.request$ **Output:** $nc: NewsCollection$ **DomPre:** $nc.state = req\_initialized$ **DomPost:** $nc.state = news\_received$ **ReqPre:** $r.details.keyword \neq$ "" $\wedge\, r.details.date \neq null \wedge$ **TrigPre:** $Collect(keyword, date)$ **ReqPost:** $\forall(n \in nc.news \mid n.keyword = r.details.keyword \vee$ $n.date = r.details.date \wedge n.text \neq null \wedge$ $(\exists\, i \in n.images)) \wedge (\exists\, n \in nc.news)$
Provide Content	**Input:** $nc: NewsCollection$ **ReqPre:** $(\exists\, n \in nc.news)$ **TrigPre:** $nc.state = news\_received$ **ReqPost:** $ShowNews(news) \wedge nc.news = news$

Table 3.2: Definition of the operations of the service platform.

For example, operation *Find Graphical Content*, shown above moves the system from a state in which the news that have to be collected are initialized (*DomPre*) to a state in which a set of suitable news is available (*DomPost*). This operation is triggered as soon as the collection of news matching provided keywords and date is started (*TrigPre*). The effect of this operation is to collect a set of news that match the date and keyword provided by the user (*ReqPost*) and contain images.

3.3 Fuzzy goals

As said before, KAOS only allows one to assess whether goals are fulfilled or not, and there is no way to say "how much" a goal is satisfied/violated. This may be sufficient for hard goals, but it is not satisfactory for soft goals, that can be satisfied up to a given degree, or when we need to tolerate small violations. For example, KAOS is good for goal G1.2 since we are only interested in knowing whether news are provided to the user within y time instants after they are collected.

In contrast, the use of KAOS to formalize goals G1.3.1 and G1.3.2 is not sufficient: since goal G1.3.1 is soft, we can only infer its partial satisfaction (a value between 0 and 1) from the contributing goals (i.e., goal G1.1.2). While for goal G1.3.2 we may want to tolerate "weak violations" that may happen when the end-to-end response time of the news provider is a bit greater than what we would like to tolerate. Furthermore, if we were able to track the level of satisfaction of these goals, we would be able to adjust the behaviour of the system accordingly. For example, since goals G1.3.2 and G1.4 are in conflict, one may try to find a compromise between the two.

The less critical requirements (G1.4) can be relaxed depending on the satisfaction of the other one (G1.3.2)[4] to provide viable solutions.

This is why IRENE distinguishes between *crisp* and *fuzzy* goals. The fulfilment of the former is boolean, while the latter can be satisfied up to a certain level between 0 and 1. Both types of requirements can be rendered in natural language, but the formal version of IRENE requires that crisp goals be expressed in LTL, and fuzzy goals in a fuzzy temporal language. This language allows one to state properties that, for example, must be verified "almost always", "within around t time instants", "always except for at most some x cases", and so on. These two types of goals can easily coexist: crisp goals represent firm requirements, while fuzzy goals add a flavour of flexibility. As commonly done for LTL we use the set of natural numbers as temporal domain. Crisp formulas are self-contained, that is, they cannot include any fuzzy sub-formula, while fuzzy formulas can also contain crisp sub-formulas.

The fuzzy language is based on the idea of membership function: it assigns a degree of truth to each proposition. Possible values are: absolutely true (1), absolutely false (0), or an intermediate degree of truth (a value in between 0 and 1). This document only presents some hints of the formalization of the fuzzy language mainly aimed to give an idea of how things work.

The semantics of fuzzy relational operators is shown in Figure 3.5(a), while the corresponding crisp operators are presented in Figure 3.5(b). For example, x = 0 is absolutely true only in 0, that is, the point in which the constraint is verified; it is absolutely false in all the other points. Similarly x < 0 is absolutely true in (- ∞, 0) and is absolutely false in [0, ∞). Fuzzy relational operators use a smooth function to assign a degree of satisfaction between 0 and 1 to those propositions that do not fully respect the condition, but are close to it. For example, $x \uparrow 0$ is absolutely true for the points close to 0 ([-1,1]), has a degree of truth between 0 and 1 in the points near 0 (e.g., [-4,-1) / (1,4]), and is absolutely false elsewhere. Given these definitions, crisp membership functions are a special case of the fuzzy ones and can only assume values 0 and 1. This means that fuzzy goals can be considered a generalization of crisp ones.

This language allows us to modify the definition of goal G1.4 and consider the number of adopted servers to quantify the severity of the violation. Needless to say, the highest the number is, the worse the violation is. The goal can be redefined as follows: $servers \preceq N_{MAX}$ and the membership function of the relational operator $\preceq$ is similar to the fourth function shown in Figure 3.5(a), where the domain is the number of servers and the points in which the membership function is completely satisfied are those in $(- ∞, N_MAX], while it gradually decreases its truth degree for those points in (N_MAX, ∞).

[4] The less goal G1.3.2 is satisfied, the more goal G1.4 is relaxed.

$x \uparrow 0$

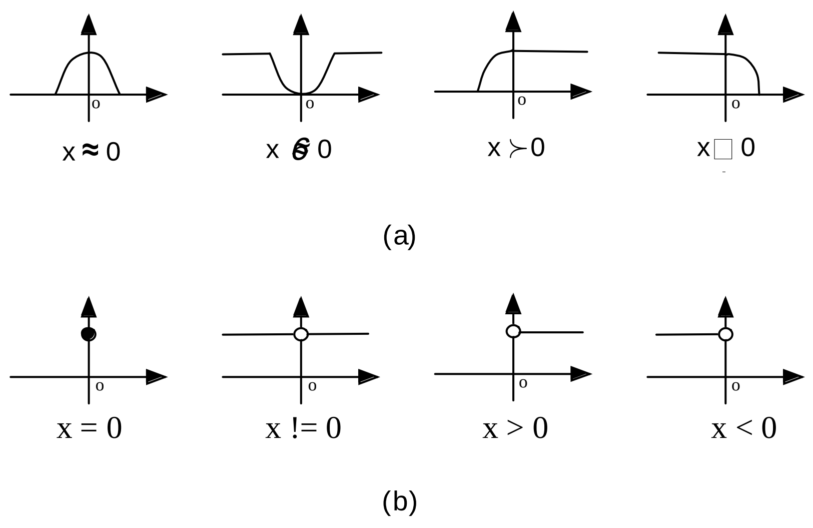


Figure 3.5: (a) Fuzzy membership functions for relational operators and (b) Crisp membership functions for the same operators.

Different semantics are available for fuzzy boolean connectives. For example, for operator 人, one can consider either the minimum of the two addends or their product. The actual choice depends on the optimism one wants to embed in the requirements.

Besides considering the fuzzy connectives and propositions, we are also interested in adding an intrinsic vagueness to temporal operators. Table 3.3 shows a brief comparison of crisp and fuzzy temporal operators. For example, for goal G1.3.2 we want to tolerate the situations when the news provider is slightly late, and it takes a bit more time to provide news. This corresponds to evaluating the satisfaction of goal G1.3.2 by taking into account also some time instants greater than the maximum delay. If this is set to t, one may use the membership function of Figure 3.6 to state that any delay between 0 and t is perfectly acceptable, while the more the delay increases, the acceptance decreases quite quickly.

Crisp op	NL expr	Fuzzy op	NL expr
G	Always	$\mathcal{G}$	Almost always
$G_{<t}$	Lasts t instants from now	$\mathcal{G}_{<t}$	Lasts hopefully t instants from now
$G_{>t}$	Always from exactly t instants	$\mathcal{G}_{>t}$	Almost always from exactly t instants
-	-	$\mathcal{G}_{-x}$	Always except at most some x cases
F	Eventually	-	-
$F_{<t}$	Within t	$\mathcal{F}_{<t}$	Almost within t instants
$F_{>t}$	Eventually from exactly t instants	-	-
U	Until	$\mathcal{U}$	Almost Until
-	-	$\mathcal{L}_{<t}$	Lasts hopefully at most t instants from now
-	-	$\mathcal{W}_{<t}$	Almost within around t instants from now

Table 3.3: Temporal operators.

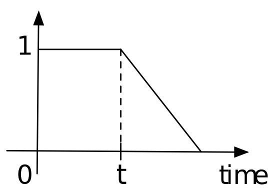


Figure 3.6: Example membership function.

According to this semantics, goal G1.3.2 is redefined as follows:

$$ReceiveRequest(keyword, date, user, pwd) \Rightarrow \mathcal{W}_{t<RT_{MAX}}\ ShowNews(news)$$

Every time the formalization of a goal introduces a new membership function, we must define its shape by taking into account the preferences of the different stakeholders. Our approach requires that membership functions be limited and continuous, and, in most of the cases, they have a trapezoidal shape (that can also degenerate into triangles). For each of these trapezoidal functions, we must define the domain ([d,D]) over which it can assume values between 0 and 1. We must also specify two key points in the domain: the minimum (m) and maximum (M) values for which the corresponding crisp formula would be true (i.e., the upper parallel side of the trapezoid).

The user can tune the severity of these membership functions by expressing the gradient of the segments that go from point (x,0), with x $\in$[d,m), to point (m,1), and from point (M,1) to (y,0), with y $\in$ (M,D]. Non-expert users can just specify a severity level among: slow, medium, and high when specifying goals.

3.4 Adaptation goals

During requirements elicitation, one must also define how the system-to-be should adapt itself at runtime. For example if goal G1.2.1 is violated, since provided news are not coherent with those requested by the user, the platform-to-be should select a new agent to perform operation *Find Graphical Content* to try to meet the goal. We must also restore the platform in a state in which news have been not collected yet but a request has been already issued. To this aim, we define *adaptation goals*, which state how the system should adapt itself by applying suitable actions. Each goal is associated with:

- An event that triggers its execution (e.g., goal G1.2.1 is violated). A trigger expresses a constraint on the satisfaction of a leaf goal and activates the execution of the adaptation goal if the corresponding conditions are satisfied too.
- A condition for its actual activation (there are too many incoherent news). A condition specifies properties of the system (e.g., satisfaction levels of conventional goals, adaptation goals already performed) or the environment (e.g., domain assumptions) that must be true to activate an adaptation goal.
- The actual adaption plan. A plan defines what should be done on the system (and on the goal model) to satisfy, or at least increase the satisfaction of, the goal that is violated.

Again, IRENE allows the user to embed adaptation capabilities in two different ways. Informally, through the use of the natural language, the user can simple provide comments and suggestions for the designers of the system. If specified more rigorously, triggers and conditions can be properly transformed into monitoring directives for the actual platform, while adaptation plans define the skeleton of the adaptation/recovery solutions.

In both cases, adaptations can be conceived as activities on the system, without modifying the goal model, that is, without modifying the requirements behind the system, and/or as changes in the model, which means that there are new/different requirements that must taken into account.

Since the identification of all possible situations in which the system must adapt could be very time consuming, and thus expensive, and in some cases all these possible situations cannot be foreseen, IRENE let the user free to distinguish between:

- Foreseen adaptations. These cases are those the user knows completely and s/he wants to fully specify them. This means that they can be either those that are considered to be more important, or the only ones the user is aware off. As the name suggests, the adaptation goals that deal with these situations are fully specified in the way the user prefers.
- Partial adaptations. These cases are more placeholders for possible problems than complete solutions. The user knows the situation(s) in which there could be problems, and s/he wants to highlight them, but there is not enough knowledge, or resources, to specify corrective actions completely. Clearly, if the user decided to only use natural language specifications, this case could be avoid since very light definitions may "solve" the problem, at least from a syntactical point of

view. With the full version of IRENE, which will require detailed specification, the user must find the proper balance between effort spent and provided details.
- Default adaptations. These are the situations the user cannot really foresee. There is no idea of when and how they may manifest, but there is anyway the interest of saying what the system should do. Since the user does not know when and how problems may arise, s/he can only identify some kind of general-purpose solutions to keep the system at least in a consistent ---and safe--- state also in these cases. These adaptations can be seen as "default" solutions: there is nothing specific, but the system must try to keep itself alive. Obviously, there are no triggers associated with these adaptations, but the user can only add conditions to state some (general) conditions that must hold before applying the adaptation.

3.4.1 Adaptation actions

Besides the well-known natural language, IRENE allows the use to state adaptation plans in terms of some predefined actions. If we consider the system level (i.e., the goal model is not changed), IRENE provides two basic operations:

- *perform(g, trans/o, trans),* resumes the execution to the first operation of goal g that is performed (in the first case), or to operation o, directly (in the second case). The second parameter (trans) indicates whether g or o already belong to the goal model or they must be added explicitly. In the former case, the execution must be resumed at the beginning of g/o starts. In the latter case, g/o must be performed on the fly, without being added to the model.
- *substitute(a1, a2/a1, a2, o)*, substitutes agent a1 with a2 for all operations performed by a1 (in the first case), or only when a1 executes operation o (in the second case). We assume that both a1 and a2 have been already defined in the object model.

If we work at goal level, the actions are:

- *add(g1, g2[, ref])*, adds conventional goal g1 among the refining goals of g2. If g2 is not refined by any goal, this action also adds a specific refinement (ref) to g2.
- *remove(g),* removes convention goal g, and if g's parent goal (g1) remain with no descendants after the removal of g, it also removes the refinement link.
- *add/remove(ag),* adds/removes adaptation goal ag.
- *modify(g, def)*, modifies the definition of a leaf goal g with another one (def).
- *relax(g1, g2/g1, perc)*, relaxes one of the membership functions used in the definition of goal g1, by reducing its slope depending on the satisfaction of g2 (in the first case) or according to a given percentage perc (in the second case). Note that if goal g1 contains more than one membership function in its definition, we assume that this operation relaxes the outermost.
- *add(o, g)*, adds operation o to goal g.
- *remove(o)*, removes operation o.
- *substitute(g1, g2 / op1, op2)*, substitutes conventional goal g1 with g2 (in the first case), or substitutes operation op1 with op2 (in the second case).
- *add/remove(obj)*, adds/removes an object obj. It can be an entity, an event, an association, or an agent.

Note that each adaptation can be performed up to a given number of times, configured by the user.

Adaptation actions can be tuned according to the satisfaction level of conventional goals. For example, if we deal with critical goals, or if a goal largely deviates from the desired objective, an adaptation that keeps the goal model "as-is" is preferred to another one that relaxes some membership functions or makes the goal model less restrictive. In contrast, in those situations in which a goal cannot be completely satisfied (i.e., its satisfaction is always under a certain threshold), we must replace a goal with another one or relax its membership function.

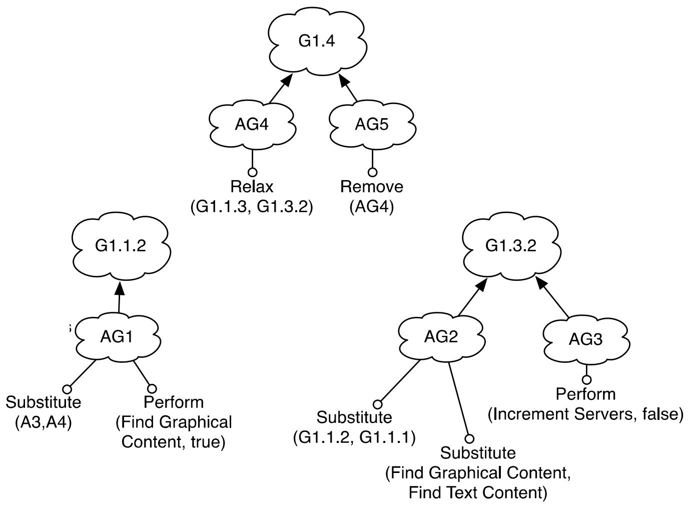

G1.4
AG4
AG5
Relax (G1.1.3, G1.3.2)
Remove (AG4)
G1.1.2
G1.3.2
AG1
AG2
AG3
Substitute (A3,A4)
Perform (Find Graphical Content, true)
Substitute (G1.1.2, G1.1.1)
Substitute (Find Graphical Content, Find Text Content)
Perform (Increment Servers, false)

Figure 3.7: Adaptation goals for the News provider.

Some adaptations may require to be applied synchronously, at the execution point in which the trigger and conditions are true. This can be useful to cope with failures in the exact moment in which they take place. Otherwise, adaptations are applied asynchronously.

Adaptation goals may conflict the same ways as conventional goals do. Conflicting goals are in general associated with different priorities (with the most critical ones having the highest priority).

The adaptation goals envisioned for our example are shown in Figure 3.7. Adaptation goal AG1 is triggered when goal G1.1.2 is violated (i.e., its satisfaction is less than 1). AG1 is performed when the satisfaction of goal G1.1.2 is less than 0.7 (condition) and it comprises two basic actions: it changes the agent that performs operation *Find Graphical Content* (A3), with another one (A4) and repeats the execution of the same operation. AG1 is synchronous since adaptation must be applied when the trigger and conditions are verified, since, in this case, we aim to cope with temporal failures of agent A3. Adaptation goals AG2 and AG3 are triggered when goal G1.3.2 is violated (trigger). In particular, they are applied when the average value of the end-to-end response time of the news provider is greater than 3 s (condition). AG2 switches to text news (i.e., it substitutes goal G1.1.2 with goal G1.1.1 and operation *Find Graphical Content* with *Find Text Content*). AG3 is applied if AG2 cannot be performed anymore. It still tries to enforce the satisfaction of goal G1.3.2, by incrementing the number of servers in the pool according to the severity of violation (it performs operation *Increment Servers*). Agent A5 can perform operation *Increment Servers* to modify the number of servers (entity *servers*, external to the system) used by the load balancer.

Adaptation goals AG1 is in conflict with both AG2 and AG3, since they try to enforce conflicting goals. According to our policy, AG2 and AG3 are triggered first, since they are associated with goal G1.3.2, which has higher priority (p= 5) than G1.1.2 (p = 3).

Finally, when goal G1.4 is violated (trigger) adaptation goals AG4 and AG5 can be applied. AG4 is executed only if the satisfaction level of G1.4 is between 0.5 and 0.8. It relaxes the membership function used to represent goal G1.3.2 depending on the satisfaction level of goal G1.4. AG5 is applied only if the satisfaction level of goal G1.4 is less than 0.5: it deletes adaptation goal AG4.

In this example partial adaptations may be necessary to handle context changes or situations in which foreseen adaptations are not effective. An example of context information for the news provider can be the device through which the customer issues a request. For example, news are in general viewed from a laptop, but they can be also requested from a smartphone. In this last case, the news provider must offer news with the right resolution. Partial adaptations are also needed when the response time of the news provider is too high (goal G1.3.2 is not satisfied) and all possible adaptation goals (AG2, AG3) have been already performed. In this case, a possible solution can send news via SMS.

The adoption of foreseen adaptations to handle these situations would not be a convenient choice. In fact, in the first case, the resolution suitable for a specific device may be unknown at design time, or the provider may prefer to add an adaptation only when necessary (e.g., the global satisfaction of the root goal, G1, is under a certain threshold). In the second case, one may prefer to embed a new adaptation only when the adaptation goals (AG2, AG3) associated with goal G1.3.2 fail and cannot be performed anymore. Furthermore, the adoption of a service that sends SMS may be expensive, and the provider might prefer to stipulate a contract, only when it is strictly necessary.

Default adaptations are used to embed some general-purpose solutions. For example, the user can add suitable adaptation actions when the satisfaction of goal G1 is under 0.3. The solution is to perform operation *Change Resolution* on the fly, before invoking operation *Show News*. We also need to add object Device that encapsulate necessary context information and agent A6 in charge of performing operation *Change Resolution*.

3.5 Variability

Even if IRENE provides special-purpose constructs to support variability, the user can actual specify it in two orthogonal ways. Analysts can exploit the OR decomposition provided by the goal model to identify alternative solutions: functional goals, soft goals, and also adaptation goals. This way of working is quite natural when the user wants to emphasise the differences (alternatives) at goal level, but the actual variability of the system-to-be remains quite hidden. Working at goal level often imposes a too wide granularity, and the precision (and amount of information) is limited. This is mainly because variability is not addressed explicitly in the model, but it is inferred as a consequence on designed models.

In contrast, IRENE considers variability a first class citizen of the requirements elicitation approach and provides suitable means to address it. In this first version, the user can add variability annotations to each element of the model. We think that this option is not an alternative to the use of OR refinements, but the two different solutions can easily coexist and should address variability in two different, but complementary, ways. Decompositions in the goal model should concentrate on what provided by the system, which would not really define system variants, but actually alternatives that "all" variants should provide. In contrast, annotations let the analyst concentrate on the differences among alternative configurations, customizations, deployments, or versions of the system. IRENE does not forbid the use of OR decompositions as a means to materialize variability, but we think this would not be enough, and users should be warned about the particular use of the feature. Again, this is where annotations can be useful to help the user understand the actual meaning of the different constructs.

The current version of IRENE is quite liberal as far as annotations are concerned. They can be used to specify:

- Some particular characteristics of a goal. One may easily think of different levels of qualities (soft goals) or variations of intended functionality (for example, different ways of showing news);
- Differences in the implementation of an operation. For example, requested could be collected in different ways;
- Alternative values for some parameters of an entity. For example, one may want to specify different (alternative) resolutions for the images that can be retrieved;
- Different responsibilities in the goal model. For example, goal A1 could be responsible for different operations under specific conditions;
- Expected behaviours of an actor. For example, again A1 may provide collected request in different ways according to different modalities;
- The satisfaction of fuzzy goals in particular contexts. For example, the membership function may change according to the different classes of users;
- Variants in the adaptation goals. For example, the adaptation may provide variants in its behaviour.

Note that some of these elements could also be added as explicit alternatives in the goal model directly, but this would easily increase the complexity of the design models with many alternatives and dependences. Moreover, these models would be tangled, and they would also hide the dependencies among the different alternatives. IRENE provides a simple, but effective, means to allow the user state the elements in the model the way s/he prefers, thus making possible dependencies become explicit. The next version of IRENE will be more prescriptive on how to write the different annotations and also on how to relate them.

3.6 IRET

This section presents the first version of the tool for the goal-based requirement engineering modelling, named IRET (IREne Toolset). The tool has been developed following a model-driven approach, and it is available as a set of plug-ins for the Eclipse platform. The instructions for download and install the tool are available in the Appendix.

We decided to build IRET on top of the Eclipse platform for several reasons: the architecture of Eclipse is general and flexible enough to develop that kind of applications (visual editors); additionally, among the projects related to Eclipse there are a set of tools and frameworks useful for model-driven development. Finally, Eclipse provides a system to build general-purpose applications, the Rich Client Platform, allowing us to build and release IRET as stand-alone application.

The following sections briefly describe the development of IRET, present the graphical elements we defined for the model used in this release, introduce the editor graphical, and provide some instructions for the creation and the management of IRENE models.

3.6.1 IRET and Eclipse

Eclipse provides a set of tools and frameworks to help the development following a model-driven approach[5]. In our development, we considered mainly two frameworks: the Eclipse Modeling Framework[6] (EMF) and the Graphical Modeling Framework[7] (GMF). A general schema representing the development process of IRET is available in Figure 3.8.

EMF is a set of components that help developers define the model (the domain model) and generate related code. The model can be generated starting from annotated Java classes, XML documents or UML models. Alternatively it is possible to define the model from scratch by using a graphical tool offered by EMF. The domain model is stored in an XML file with `ecore` extension; generally, this file can be used to generate the Java code of the model, and also as input for other Eclipse modelling tools to perform a wide range of operations, such as model to model transformations, model checking, construction of graphical editors, and so on. For our purposes, we used the domain model to generate two plugins:

[5] http://www.eclipse.org/modeling/

[6] http://www.eclipse.org/modeling/emf/

[7] http://www.eclipse.org/modeling/gmp/

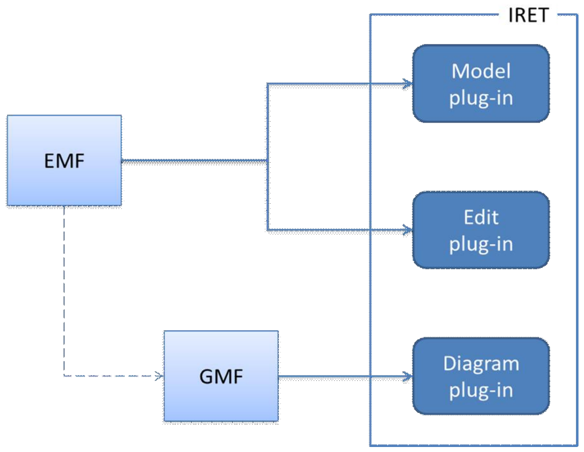


Figure 3.8: Development process of IRET.

- Model plug-in contains the Java classes of the elements composing the domain model and a factory class;
- Edit plug-in contains the base classes useful for building the editor, such as label providers, property sheet generators and a command framework to manage the editing of the model (with undo and redo features).

GMF provides a set of tools to build graphical editors based on EMF models. Figure 3.9 represents the GMF development process.

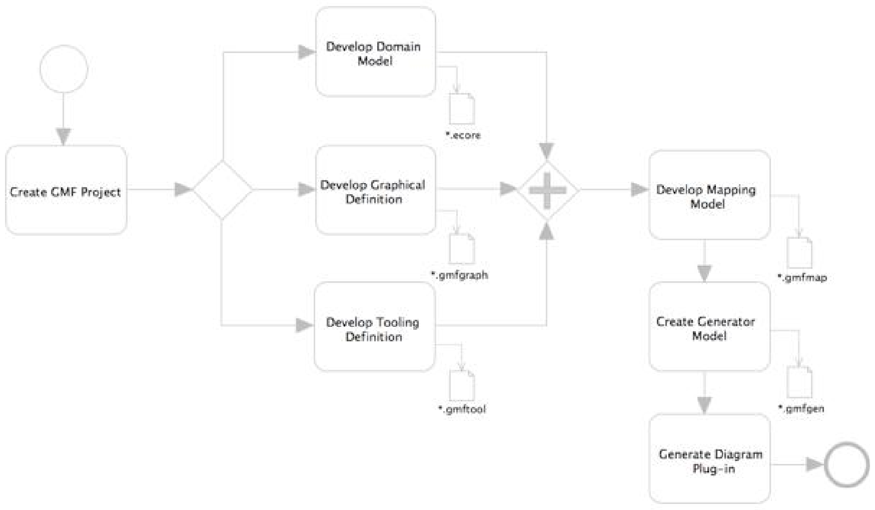


Figure 3.9: Development with EMF and GMF[8]

As shown in the figure, after the creation of the GMF project, one must develop three files that will contribute to the definition of the graphical editor:

- the Domain Model: the `ecore` file describing the model, built with EMF as explained above;
- the Graphical Definition: a XML file with `gmfgraph` extension, that defines which objects will be visualized in the graphical editor (nodes, labels, links) and their graphical aspects (shape, color, etc.)

[8] Image from: http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1

- the Tooling Definition: a XML file with extension `gmftool` containing the list of the elements that will be visualized in the palettes of the editor.

After the definition of the model, the graphical elements and the palettes, one must define a relation model to specify the links between the components of the three files. In other words, each graphical element of the editor should be related to a concept of the domain model and, optionally, to an element of the palette (for example to create new instances). The output of this step is the production of a XML file with extension `gmfmap`.

The mapping file is the input to create the generator model, an XML file (with `gmfgen` extension) containing all the required information for the generation of the graphical editor. This file contains not only the mapping between the domain model and the elements of the editor, but also other information, such as the menus, the parsers, the editor metadata (the author, the version) and so on.

Finally, the generator model supports the generation of the Java code: the output is the third plugin shown in Figure 3.8, the Diagram plugin. It depends on the Model and Edit plugins and provides the classes to have a complete editor: the graphical elements, the classes for the creation and the modification of the model (through invocation of the Editor plugin), the classes to serialize created models in XML files.

Finally, we modified the three plugins to customize the behaviour of the editor. Some examples of our modifications are the addition of images to the graphical elements, the creation of a dedicated perspective for the editor, and the definition of additional features to be visualized in the property sheets for particular model elements.

3.6.2 IRET elements

As already said, IRET implements IRENE through EMF. An EMF model is pretty similar to a UML class model: there are classes, representing the concepts of the model, with attributes and methods. Classes are connected through different kinds of relations (inheritance, user-defined relation, etc.). Additionally it is possible to insert rules using the Object Constraint Language (OCL).

Due to the fact that a detailed description of the meaning of the IRENE elements is supplied in previous sections, here we briefly present the EMF model without going in depth in the semantics of the components. For the sake of clarity we represent the main elements of the model in a couple of class diagrams. The first one (Figure 3.10) shows the part of the model related to goals and responsibilities.

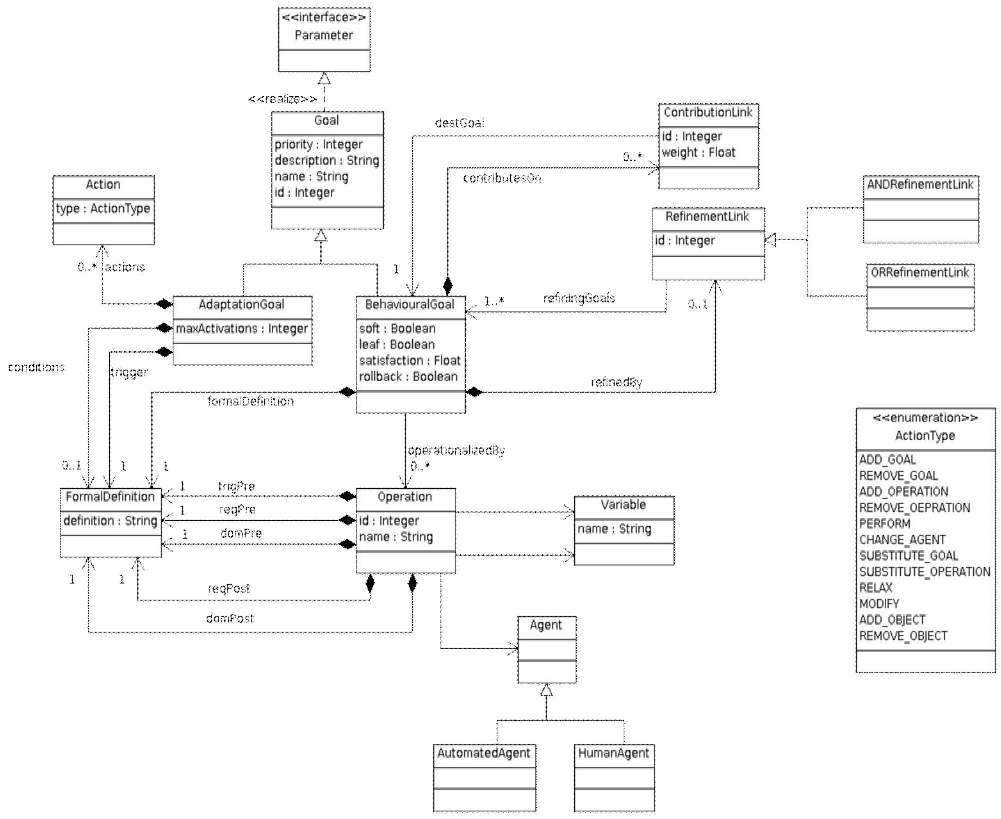


Figure 3.10: EMF model of IRENE - Goal and responsabilites.

The main element in the diagram is class Goal; goals can be specialized in behavioural goals (as defined in KAOS) and in adaptation goals. Behavioural goals are related through refinement and contribution links: the former allows one to split a goal in a combination of simpler goals, while the latter indicates how much a sub-goal contributes to the satisfaction of the super-goal.

Additionally, leaf goals (goals without refinements) are associated with one or more operations that decompose the goal; the constraint that only leaf goals can be “operationalized by” operations is expressed though a rule.

Operations are then supplied by agents, which can be humans or automated. The relation of responsibility, which links agents to goals, is expressed as composition of relations operationalizedBy and opeart:

$$responsibleOf \equiv opeart \circ operationalizedBy$$

Figure 3.11 shows the object model: the main class of this diagram is class Object, characterized by an ID and a name. Object has a set of elements that inherit from it: as explained above, there are Events (instantaneous actions), Entities (informative content providers), Agents (goals responsible elements) and, indirectly, Associations (relations between objects).

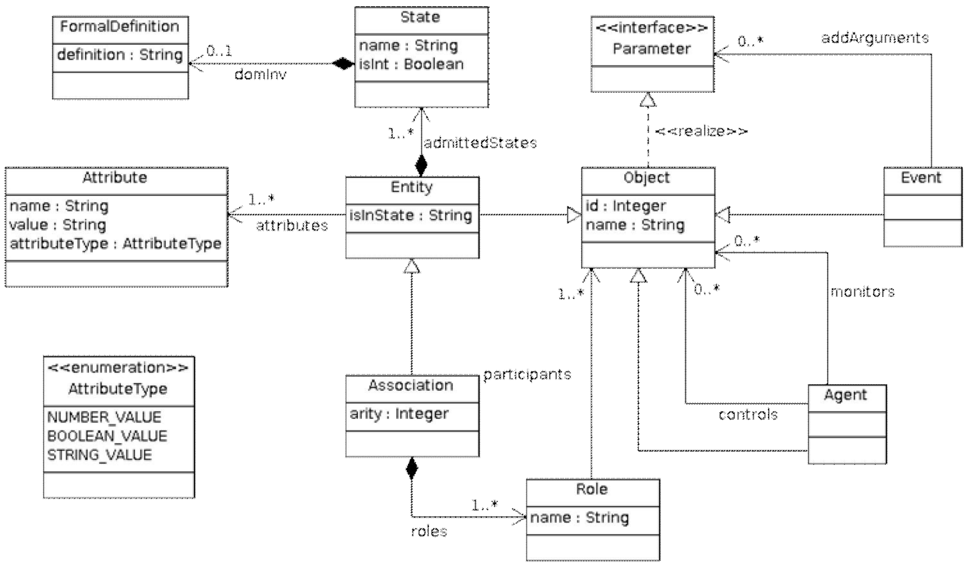


Figure 3.11: EMF model of IRENE – Objects.

Finally, Table 3.4 introduces the graphical notation used in the GUI of IRET to identify the different elements.

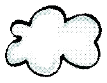
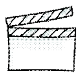
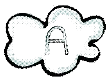
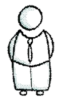
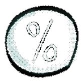
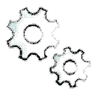
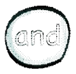
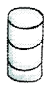
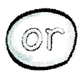
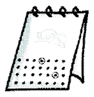
IRENE Element	IRET Representation	IRENE Element	IRET Representation
Behavioural goal		Operation	
Adaptation goal		Human agent	
Contribution rink		Automated agent	
AND refinement link		Entity	
OR refinement link		Event	

Table 3.4: Elements used in IRET GUI.

3.6.3 Graphical interface

IRET provides a full-fledged graphical interface/editor that offers all the elements in IRENE to let the user design complete and consistent requirements elicitation models. This perspective, according to the Eclipse jargon, can be selected from the perspective list of Eclipse (Window → Open perspective → Others…).

Figure 3.12 shows a screenshot of the perspective with four main components: Main panel, Project Explorer, Properties and Outline.

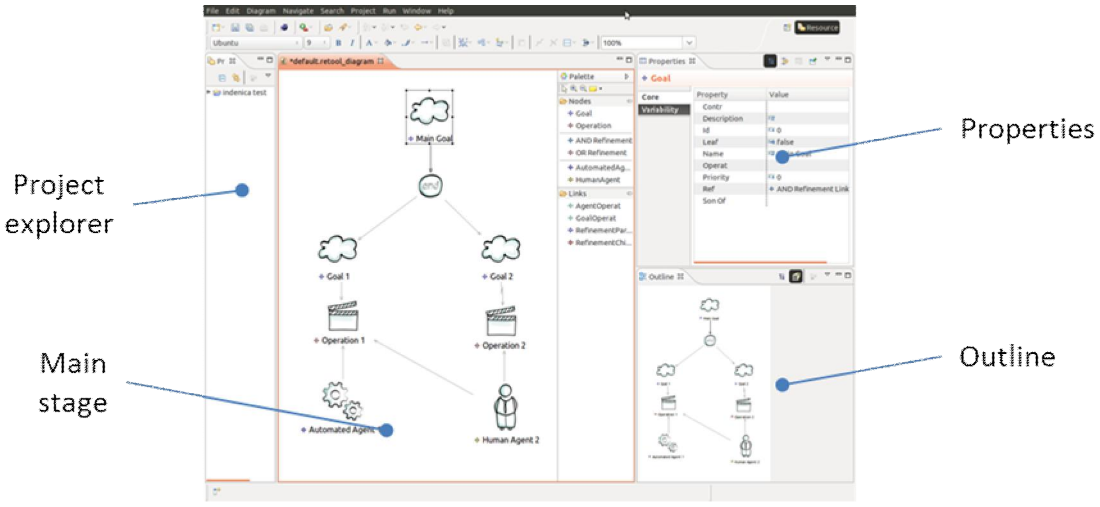


Figure 3.12: Screenshot of the application.

The user designs goal models in the main panel. This panel, Figure 3.13, is further split in two parts: a Palette, where the user can select the different elements and a Drawing area, to draw the model.

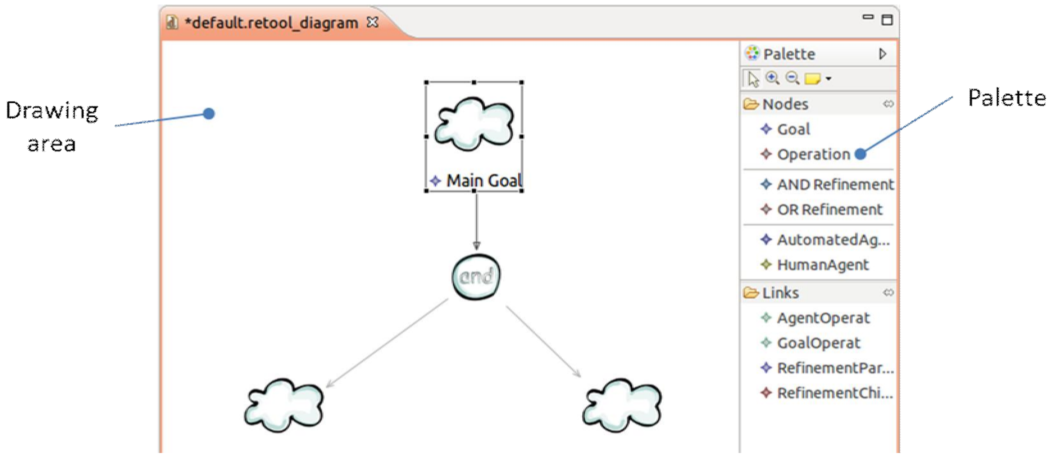
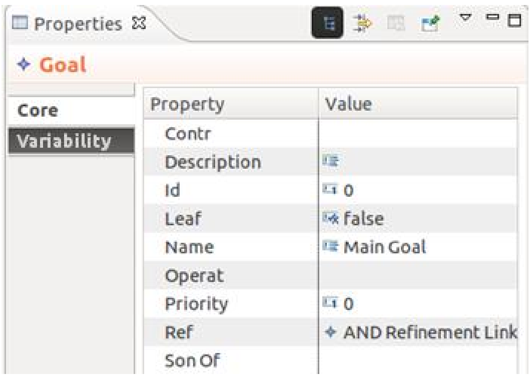


Figure 3.13: Main stage.

The management of existing elements is done through the Property panel (Figure 3.14). Each object (both nodes and links) has a set of attributes, as defined in the EMF domain model (described above). Attributes are typed and they are distributed in one or more tabs: the disposition varies dynamically depending on the element: for example operations have additional tabs for the definition of preconditions and effects.


Properties
Goal
Core
Variability

Property	Value
Contr	
Description	
Id	0
Leaf	false
Name	Main Goal
Operat	
Priority	0
Ref	AND Refinement Link
Son Of	

Figure 3.14: Propeties.

With complex models, the amount of existing elements in the Drawing area can be very high, thus affecting the readability and ease of navigation of the model. To cope with this problem, the perspective offers the Outline panel. The panel shows a thumbnail of the whole model, with a rectangle representing the area shown in the Drawing area (an example in Figure 3.15). The user can move the rectangle to select the area of interest and visualize it in the Main panel.

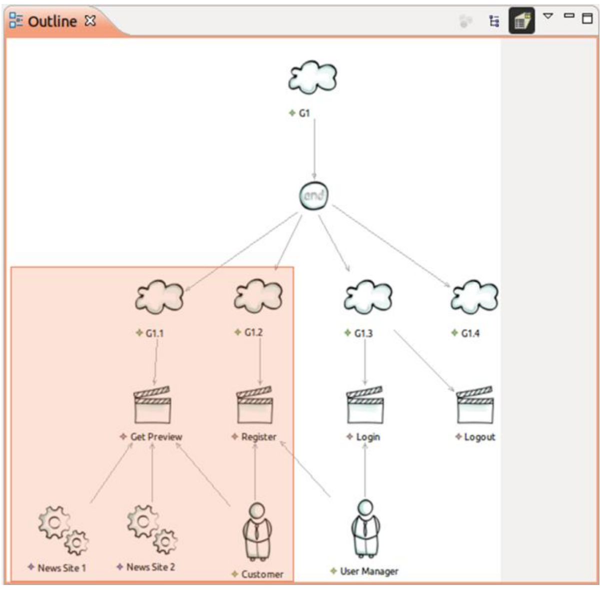


Figure 3.15: Outline

The last component of the interface of IRET is the Project explorer (Figure 3.16).

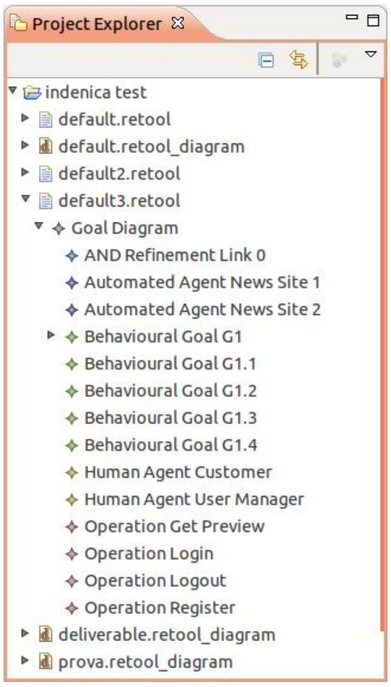


Figure 3.16: Project explorer

On the one hand, this panel allows the user to view the list of the existing projects and the files composing them; on the other hand, the user can see the list of the elements in each IRET file. For each element, it shows the class of the element (with the relative icon) and the name.

3.6.4 Creating and saving models

Each model should be related to a project. It means that to create a model, at least a project should exist in the Eclipse workspace.

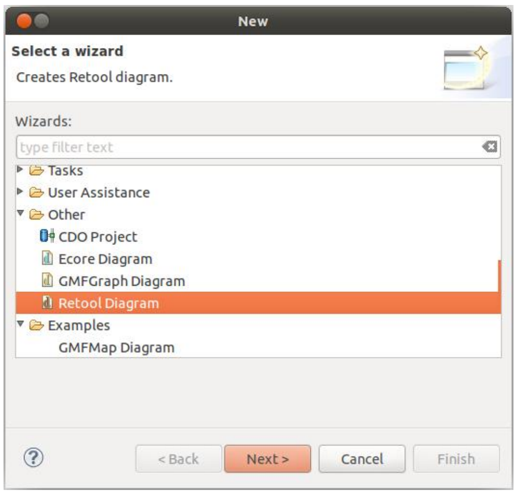
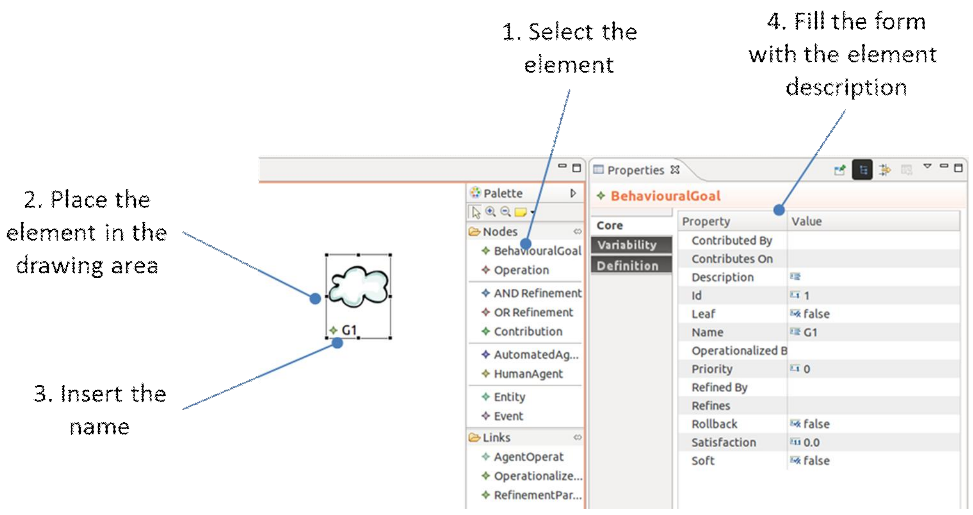


Figure 3.17: Creation of a new model.

New models can be created by selecting IRET Diagram in the New dialog (File → New... → Other...), as shown in Figure 3.17, by pressing the Next button and by following the wizard procedure. For every model, the tool creates two files: one with extension `iret` and one with extension `iret_diagram`. The first file contains the information about the model; the second contains the graphical description about the elements of the model. When a new model is created, the tool automatically opens it in the Main panel.

Then the user can start designing the model and insert new elements as shown in Figure 3.18: s/he selects the element of interest from the Palette and then places it into the model.



Property	Value
Contributed By	
Contributes On	
Description	
Id	1
Leaf	false
Name	G1
Operationalized B	
Priority	0
Refined By	
Refines	
Rollback	false
Satisfaction	0.0
Soft	false

Figure 3.18: Insertion of a new element in the model.

When positioned on the canvas, the user can add the name of the element, then visualized as label. Finally, the user can insert the description of the node in the Property view.

To insert relations (Figure 3.19) between nodes is similar to what described above: first the user selects the type of relation among the ones available in the palette, and then s/he drags the mouse from the starting element to the ending one to create the relation between them. A set of descriptors in the Property panel allows the user to modify the attributes of the relation.

Note that the editor verifies whether the relation can be set for the two selected elements: if the user tried to draw an invalid link, the editor would deny its creation. Examples of these situations are the incompatibility between the type of the source element and the domain type of the relation, or the existence of constraints that would be violated with the new link (e.g. a goal cannot be a refinement of itself).

The deletion of elements (nodes or links) from the model can be done through the “Delete from Model” command available in the contextual menu of each element (enabled by right clicking on an element).

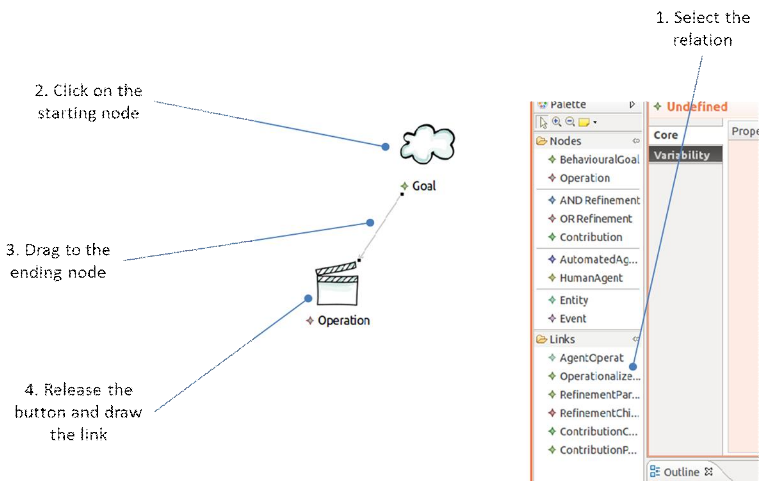


Figure 3.19: Insertion of a new relation in the model.

Models are saved by using the Save command (File → Save): as explained before, models are stored in two XML files, with extensions `iret` and `iret_diagram`. The first file contains the list of existing elements, the relations between them, and the values of the properties of each element, as reported in the following sample:

```
…
  <hasBehaviouralGoal name="G1.1" refines="//@hasRLink.0" operation-
alizedBy="//@hasOper.0"/>
  <hasBehaviouralGoal name="G1.2" refines="//@hasRLink.0" operation-
alizedBy="//@hasOper.1"/>
  <hasBehaviouralGoal name="G1.3" refines="//@hasRLink.0" operation-
alizedBy="//@hasOper.2 //@hasOper.3"/>
  <hasBehaviouralGoal name="G1.4" refines="//@hasRLink.0"/>
…
```

The second file contains the graphical description of the elements of the model: the position in the Drawing area, the font to be used for the visualization of the name and so on.

```
…
  <children xmi:type="notation:Shape"
xmi:id="_z5goc0UeEeCbe9VuGRQQ9w" type="2013" fontName="Arial">
    <children xmi:type="notation:DecorationNode"
xmi:id="_z5ido0UeEeCbe9VuGRQQ9w" type="5011"/>
    <element xmi:type="retool:BehaviouralGoal"
href="default3.retool#//@hasBehaviouralGoal.0"/>
    <layoutConstraint xmi:type="notation:Bounds"
xmi:id="_z5goceUeEeCbe9VuGRQQ9w" x="440" y="20"/>
  </children>
…
```

Since IRET is supposed to be the first tool of a chain, the distinction between the description of the model and its graphical representation is very useful: this way the components requiring as input the output of IRET could only receive the `iret` file, and ignore the `iret_diagram` one.

4 User Centred Requirements Engineering

4.1 Motivation

Each time one tries to raise the whole set of requirements from all relevant stakeholders you are faced with a bunch of challenges.

Apart from the difficulty to identify the complete array of stakeholders, it is overconfident to assume that each single stakeholder is really aware of all aspects of requirements relevant to him in order to get a comprehensive perception of all different aspects.

Platforms and service oriented environments add additional complexity to this topic. So the clear view of the user and his perception of the system under development get even more importance.

In the 50's of the last century, two American social psychologists, Joseph Luft and Harry Ingham developed a method for visualization self and external perception when executing personality tests, called Johari Window[9].

It consists of four distinct areas of perception:

- Open: The most evident area of our consciousness. It is obvious as well to us ourselves as well to others. Thanks to this own and public awareness of this area, it is called “Arena”.
- Blind: This area is dangerous, as it covers aspects which are known to others whereas they are unknown to ourselves. Therefore this area also is called “Blind spot”.
- Hidden: The third area represents those aspects which are known to ourselves but unknown to others. It is called “Façade”. Aspects of this are may actively be concealed from the recognition of other people.
- Unknown: The last area is characterized by a total lack of knowledge by others and ourselves. Obviously this section cannot be described extensively.

[9] See http://en.wikipedia.org/wiki/Johari_window (accessed 25 May 2011)

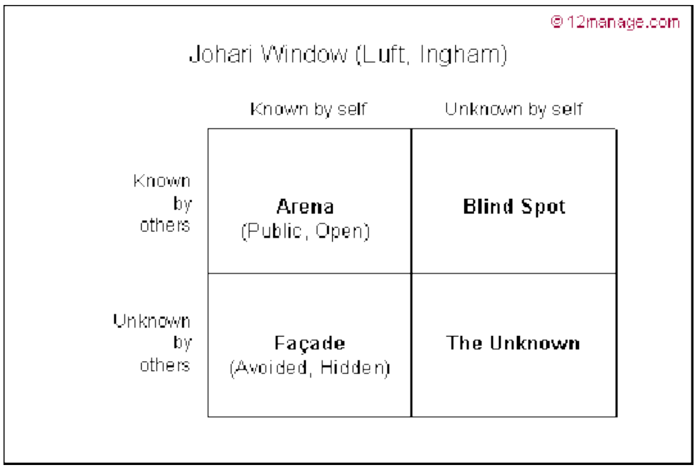


Figure 4.1: Johari-Window

Transformed to the challenge of user centred requirements engineering, the Johari Window may use as a model to describe the different facets of requirements, which are not visible in the same way to all stakeholders.

Let's consider any specific scenario:

Each user has his perception of expectations and requirements. But as seen in the Johari window model, he's not always aware of all aspects.

- For the first area (Open, "Arena") the elicitation of requirements is rather comfortable, as users and their environment know about these issues. Requirements of this group can be documented clearly. They are well-known by all stakeholders. If the relevance is similar for all stakeholder groups, these requirements are candidates for members of the core asset base.
- For the second area (Blind, "Blind spot"), elicitation requires high efforts. Often directly involved stakeholders are not aware of these requirements. One reason might be that for them requirements of this area seem to be self-evident. So other stakeholders have to be included to reveal also requirements of that area.
- For the third area (Hidden, "Façade") it may be even harder to detect these requirements, because the stakeholder – for any reason – may try to hide these requirements or at least might not admit to have them.
- For the fourth area (Unknown) it is almost impossible to detect these requirements, because no one is aware of the existence of these requirements.

Considering especially the first three areas, the need is obvious to establish a method for requirements elicitation which puts each individual user or user group into the centre of reflection.

Only the composition of all different points of view (laying all perspectives on the top of each other or beside to each other) allows to detect the overall view. On doing so, the information about

- Origin of requirement
- Priority of requirement for each stakeholder

may not get lost. An approach for ensuring this is presented in chapter 4.3.

4.2 User Centred Design

In an interview with the Digital Web Magazine [Evans, 2002] Peter Merholz and Nate Shedroff gave definitions of user centred design:

Peter Merholz:

“Contrary to common wisdom, user-centered design is not a process, but a philosophy. User-centered design requires the inclusion of a product's end-users throughout the design process.”

Nate Shedroff:

“User-Centered Design is an approach to creating experiences for people with their needs in mind. Usability is one of the primary foci but only one of several. Others include usefulness, desirability, legibility, learnability, etc. The benefits are that these experiences are often easier to use and learn; more appropriate in terms of function and use, and more compatible with existing processes.” [Evans, 2002]

A user centred model contains four dimensions of user participation in the design process (see Figure 4.2):

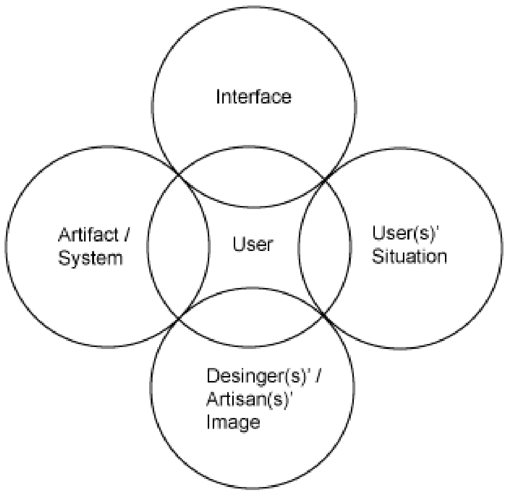


Figure 4.2 Dimensions of the User centred Model

The user-centred process places the user as an active participant in the design of the product.

ISO 13407 defines a standard for a human-centred design process with three areas:

- Human-centred design activities
- Throughout the whole lifecycle
- Interactive computer-based systems

Main activities are:

1. Specify the context of use
    - Identify users
    - Identify purpose
    - Identify circumstances
2. Specify requirements
    - Identify business requirements
    - Identify user goals
3. Create design solutions (prototypes)
    - Develop solution in iterations
4. Evaluate designs
    - Usability testing with actual users

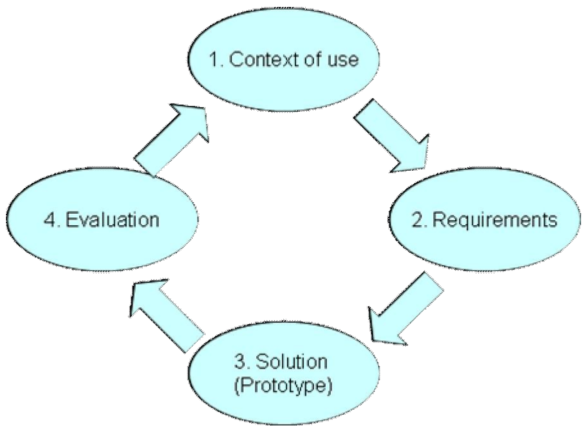


Figure 4.3 Human-centred design process

The main principles of User Centred Design are:

- Iterative Approach
- Early concentration on user requirements
- Empirical evaluation of prototype by user

Recommended Methods for User-Centred Design

- Interviews
- Contextual Inquiry
- Online survey
- Persona
  (fictional character created to represent a specific user type)
- Use Case

- Prototyping
- Usability Tests
- Review performed by experts

4.2.1 Requirements Engineering and User Centred Design

What is then „User-Centred Requirements Engineering“?

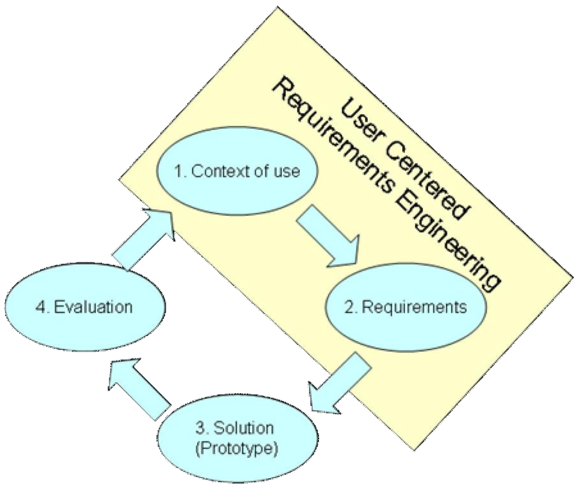


Figure 4.4 RE in the Human-centred design process

User centred Requirements Engineering may be interpreted as a well defined part within the “User-centred Design”:

The constantly recurring iteration loop the following steps:

- Context of use
- Requirements

can be identified as process steps of the “User-Centred Requirements Engineering” process.

That includes the following tasks:

- Meet with key stakeholders to set vision
- Include usability tasks in the project plan
- Assemble a multidisciplinary team to ensure complete expertise
- Develop usability goals and objectives
- Conduct field studies
- Look at competitive products
- Create user profiles
- Develop a task analysis
- Document user scenarios

- Document user performance requirements

Hongquin Sun defines “User Centric Requirements Engineering” and derives the term from “User Centric Software” [Sun 2007]. His work focuses on User Centric Software, which means software with a high user interaction

- Intensive user-computer interactions.
- Oriented to users’ responsibilities or goals.
- Usability sensitive - appropriate interface, easy operation.
- Users’ HCI satisfaction is a priority.

He refers to User Centric Development:

- The software development team includes some users.
- The user is an equal participant with the developers in making development decisions.
- The software development team develops the software using an iterative approach.
- A series of versions or prototypes of the software are developed and delivered
- With feedback from users of earlier versions of the software driving development of later versions.

4.3 User Centred Dimensions

In general, RE could be based on use cases and user stories and therefore also on involved stakeholders. One challenge in RE is to find the adequate priority for a common set of requirements, with respect to the different importance of the stakeholders.

Often RE takes the very end user as starting point and expands the application requirements from the very end user to the next involved user and so on.

Therefore RE could be seen as a more-dimensional relationship diagram of requirements. To take into account the different importance of requirements, it’s helpful to add weight factors (e.g. multi-factor analysis), so that specific requirements (e.g., security requirements) may overrule other requirements. Weight could also be assigned to certain users, according to their importance for the system. For product lines an additional challenges consists in

- defining the adequate set of requirements bound for the platform
- and the correlation of functionalities to the specific products.

For user centric RE we define three dimensions as basis:

4.3.1 User level dimension

The main questions regarding these levels are:

- Who is the user?
- Which user groups are on the level?

The sequence of levels could be defined as follows:

1. Level: end user groups- view,

2. and (n+1)th Level: user group before the end user groups, etc. "backwards" which means "in greater distance to the system under development" (See Figure 4.5).

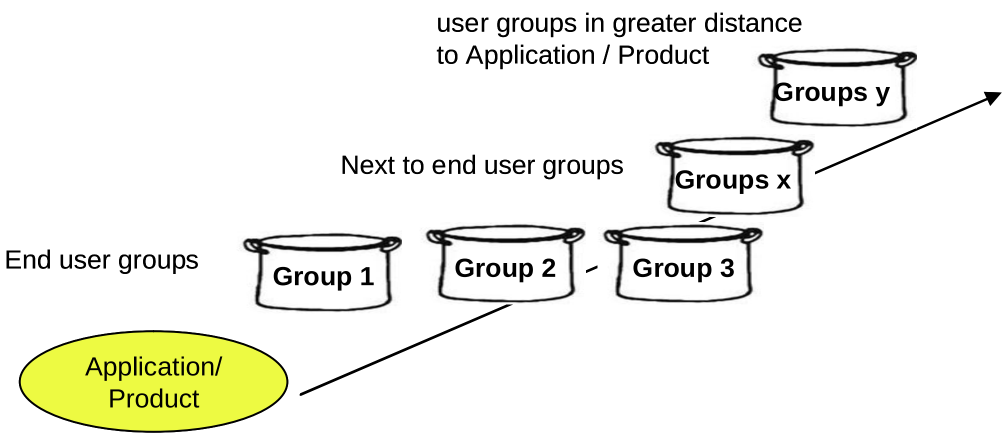


Figure 4.5 User groups in the user level dimension

4.3.2 Importance dimension

Each user has a specific priority for his requirements, but also for the topics these requirements belong to. Also the importance of these topics varies from user to user. The main questions behind these levels of the importance dimension are:

- What is the centric importance of a topic for the user groups?
- What is the most important topic for the user groups?

The symbol of "centric" could be used to visualize the importance of the topics for the user. The most important topic for the specific application is closest to the user, while the others could be placed on concentric circles around the users. See Figure 4.6

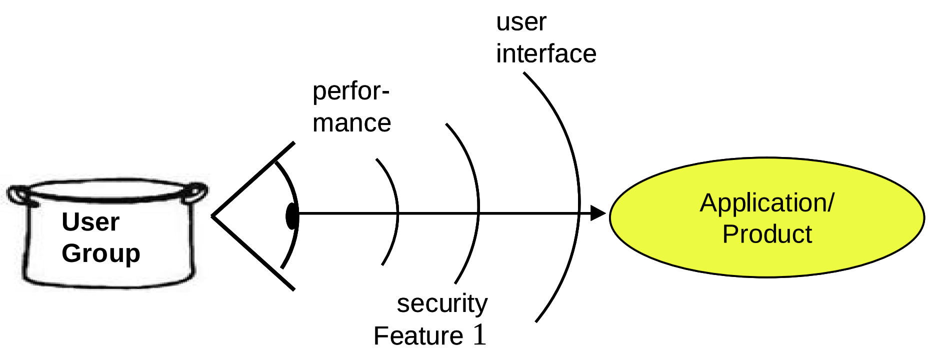


Figure 4.6 the user's perspective as input for user dimension "centric topic priority"

Hint: This order can be different for the different user groups and user groups' levels. Each user group has a different view on the application / product and rates different topics with different importance.

Samples of centric topics are: performance, user interface, different features, security, etc.

4.3.3 Requirements dimension

From the user's point of view, every user has a set of requirements for the system under development. These requirements belong to different topics. So requirements could be clustered according to the topics they belong to. For RE it is also important to get a consolidated set of requirements, to which all stakeholders are committed to.

4.3.4 Connection to the Requirements Engineering process steps

User-centered dimensions are means for structuring and categorizing users and requirements. This helps in reducing the complexity of RE. The most useful contribution will be found in the early steps of RE. The first two process steps are "requirements elicitation" and "requirements analysis".

Requirements elicitation

Requirements elicitation is the first step in the RE process.

The main challenges concerning requirements elicitation are [56]:

- Identification of relevant requirement sources
- Elicitation of existing requirements from the identified sources
- Development of new and innovative requirements

In "elicitation" the user level dimension is used for a systematic approach of collecting requirements.

- User groups on the different user levels help to identify all stakeholders.
- For each user group on each user level requirements should be collected.

The topic importance, it is also used for systematic work on elicitation.

- For each user group find the ranked sequence of topics regarding the centric importance and execute the Requirements Engineering process for each topic:
- Take user group X. Take the perspective of this user group when looking at the application / product.

  1. step: RE for the most important centric topic.

  2. and (n+1)st step: RE for the next important centric topic.

Requirements analysis

Requirements analysis is following "requirements elicitation" as second step in the RE process. The main focus is on defining a common set of requirements which could be used for further development. The main challenges concerning requirements analysis are [56,59]:

- Structure requirements
- Develop a common set of requirements
  - o Eliminate unnecessary requirements
  - o Add missing requirements
  - o Discover requirements overlaps and conflicts
  - o Analyze the cause for each conflict
  - o Resolve the conflicts
  - o Document the resolution and their rationale
- Prioritize requirements

For the prioritization we will introduce a method in chapter 4.3.5.

In platform environments one specific topic is to define the parts belonging to the platform and the parts specific to the different products. So after building the common set of requirements, as next step prioritizing and development of platform and product requirements is necessary. The user centred dimension could be used for visualizing requirements priorities given by certain user groups.

4.3.5 Mosaics of Requirements Priorities

As first step, each user group has to prioritize the common set of requirements. We draw a matrix with following axis:

- x-axis: requirements axis
  requirements are grouped together according to the topic they belong to

- y -axis : priority axis
  priority in descending order, which means 1 is highest priority, 4 is lowest priority

Each user level is given a specific colour.

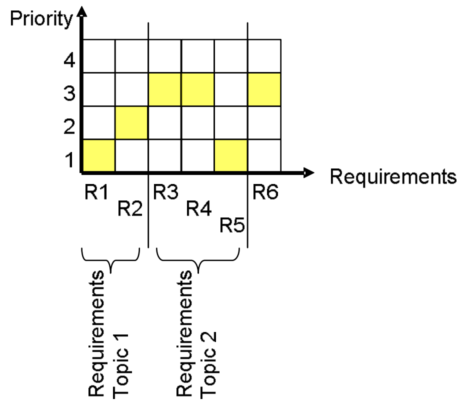


Figure 4.7 Requirements-Priority table

A coloured field is placed at the point, where the user group votes a certain priority for a requirement.

This rating is done for each user group on each user level. An example result is given in Figure 4.8. This outcome could be used for further elaboration.

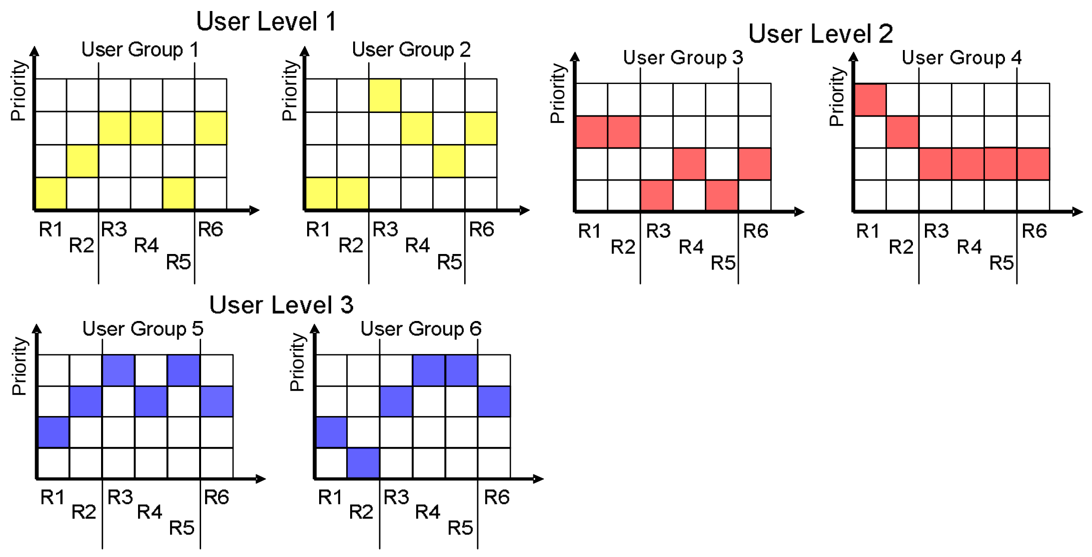


Figure 4.8 Result of voting for 3 user levels and two user groups on each user level

Priority Mosaic as combined result

Each voting of a user group is represented by a tile. User groups could be weighted. User groups with double weight get tiles in double size.

Each square is divided into as many fields, that the maximum of reachable tiles could be placed. In our example:

[double size tile (=2 single tiles) for user group1] + [1 single tile for user group 2,3,4,5,6] = (2+5) tiles = 7 tiles

For each user group their tiles are placed in the square of the priority stated for the requirement.

Figure 4.9 shows the result for the above example.

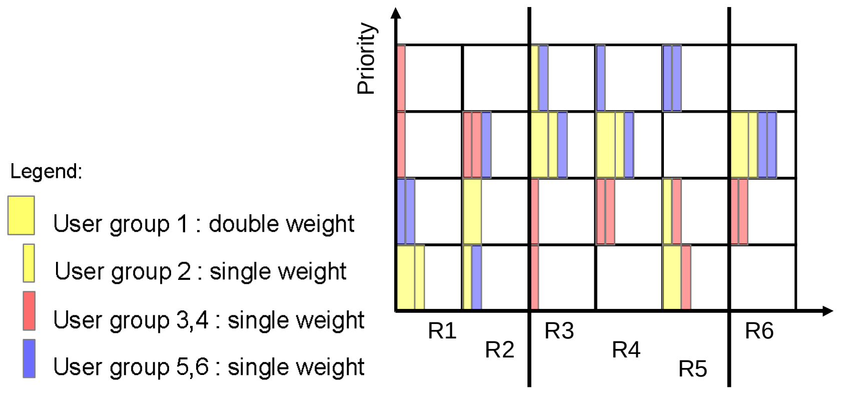


Figure 4.9 Priority mosaic

Expected result:

- Overall priority for a requirement
- What should be included in the core asset base, what is included in specific products

What we can see:

- How many user groups voted for a requirement at a certain priority
- Also information about the core asset requirements vs. product requirements is visible. For example R6 is stated from all user groups at nearly the same priority, which could be a hint for the core asset base. Whereas R5 is only important for user group three, but nor for the other user groups. This could be a candidate for a product requirement.

Conclusion:

- Visualization shows which weights are from which user groups at which priority
- Also shows areas where trade-off has to be made
- Also shows separation of product vs. platform/core asset base requirements.

4.4 Agile RE for PLE

Agile methodologies aim at making the development process more flexible, lightweight and more focused on the target desired by the customer. Bases are iterative and incremental proceedings with close collaboration with the customer. Moreover, the Manifesto for Agile Software Development reads:

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.[10]

Hereby, it is assumed that requirements cannot be fully specified at the start of the development so that close and continuous collaboration with the customers and stakeholders is vital.

One of the most common agile methodologies is the project management framework SCRUM.

The iterative and incremental approach is mapped in the *Sprints* where the output of each development Sprint is a *working increment of the product*. In SCRUM the central elements regarding RE are the SCRUM role "Product Owner" and the artefacts "Product Backlog" and "Release Plan".

The *Product Owner* (PO) is someone who has best customer and market knowledge and acts as voice of the customer to the development team. The PO has the vision of the product, can estimate its business value, ranks the features to be developed by the development team, and explains the features to them.

The PO writes the features which the customer wishes into *the Product- Backlog*. The Product Backlog contains all (product, performance, etc. related) features that the customer wants and the technical dependencies as well. The product backlog is prioritized, described in *user stories* where the high priority features are described in more detail than the low prioritized ones, and contains acceptance criteria for the user stories, their business value, the risk, rough effort estimation estimated in some unit (e.g., ideal days), and dependencies.

Then, the Product Backlog Items are assigned to in which Sprint they are going to be implemented – and this planning is kept in the *Release Plan/Release Backlog*. For each Sprint again the development team commits to implement a certain amount out of the Product Backlog Items based on the information of the Release Plan/Release Backlog. So, the Release Plan/Release Backlog has to be updated continuously according to the team's commitment, and what the team finished in the Sprint, and the respective feedback of the customer. The part of the product backlog to that the team commits for a Sprint is called *Sprint Backlog*.

[10] Source: http://agilemanifesto.org/

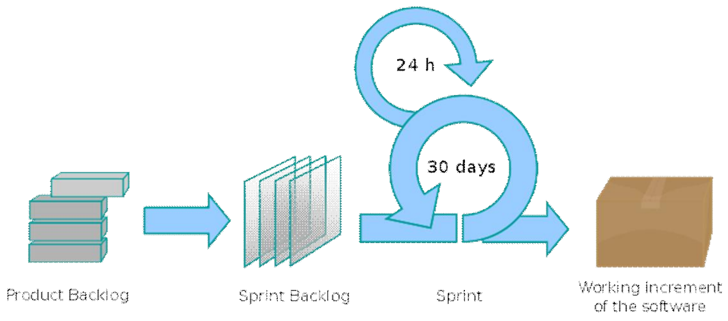
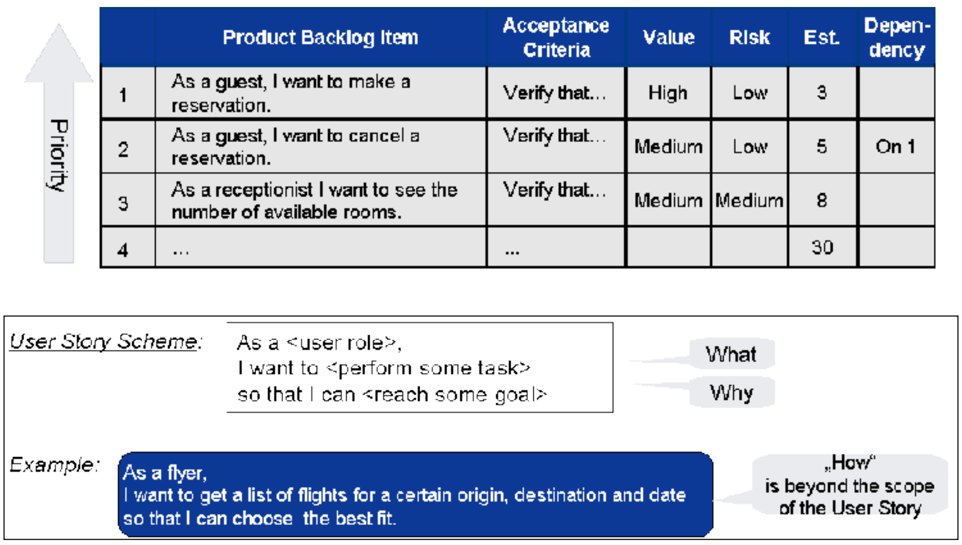


Figure 4.10: The SCRUM process[11]



	Product Backlog Item	Acceptance Criteria	Value	Risk	Est.	Depen-dency
1	As a guest, I want to make a reservation.	Verify that...	High	Low	3	
2	As a guest, I want to cancel a reservation.	Verify that...	Medium	Low	5	On 1
3	As a receptionist I want to see the number of available rooms.	Verify that...	Medium	Medium	8	
4	...	...			30	

Priority

User Story Scheme: As a <user role>,
I want to <perform some task>
so that I can <reach some goal>

What
Why

Example: As a flyer,
I want to get a list of flights for a certain origin, destination and date
so that I can choose the best fit.

Figure 4.11: Example for Product Backlog Items and a User Story

4.4.1 User centric Requirements Engineering with SCRUM

Since in agile methodologies customer collaboration is a central element, there is a lot of user feedback and user influence during the whole agile process - during the Product Owner's process part and the implementation process part as well: The user or customer is explicitly asked to give feedback to the *working increment of the product* which flows back into an update of the product backlog and release plan/release backlog. Over these artefacts, the user or customer influences the sprint backlog and the implementation as well.

With this, agile methodologies, especially SCRUM's iterative Sprints, enrich the iterative approach of the user-centred RE, described in Section 4.2.

[11] Source: http://en.wikipedia.org (accesed 25 May 2011)

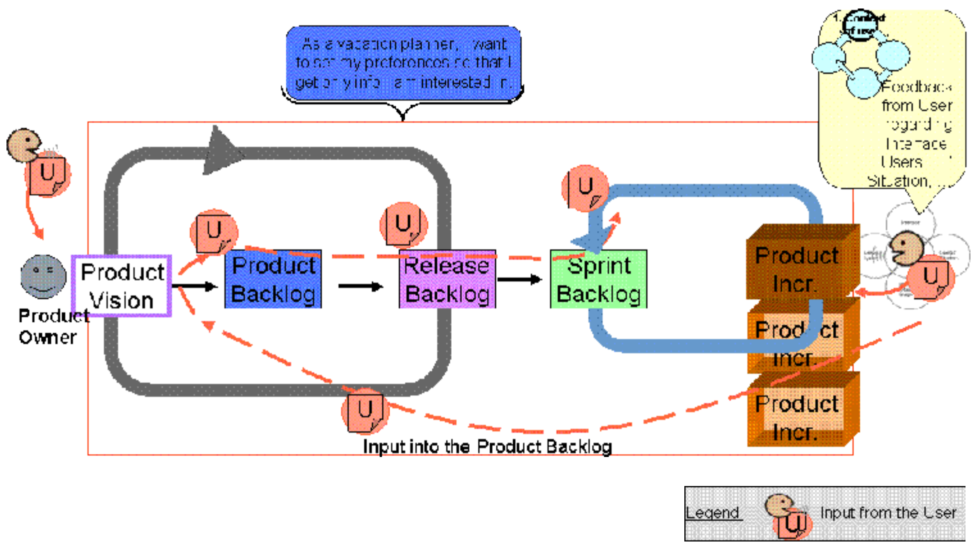

As a vacation planner, I want to set my preferences so that I get only info I am interested in.
Feedback from User regarding Interface Users Situation, ...
Product Vision
Product Owner
Product Backlog
Release Backlog
Sprint Backlog
Product Incr.
Product Incr.
Product Incr.
Input into the Product Backlog
Legend
Input from the User

Figure 4.12: Users' feedback in Agile's iterative process (enriching the iterative approach of the user-centred RE)

How agile requirements engineering can be organized, what interfaces they face and how user's feedback flows into an agile RE is described in the following.

The main decision point for the product's features in an agile development is the Product Owner (PO).

In small agile projects, there is one Product Owner who tells the (development) team what to implement.

In larger organizations however, the Product Owner (PO) does not perform requirements engineering into the detail that the developers would need to be able to implement the new product. Often, based on the input from the PO, some specialists (requirements engineers (RE's)) care for the (Software) Requirements engineering, requirements specifications and the features specifications to the necessary degree of detail for the development. They detail the Product Backlog from the PO into specific requirements for the development, care for the specific user's feedback and face so a lot of interfaces (see Figure 4.13 and Figure 4.14).

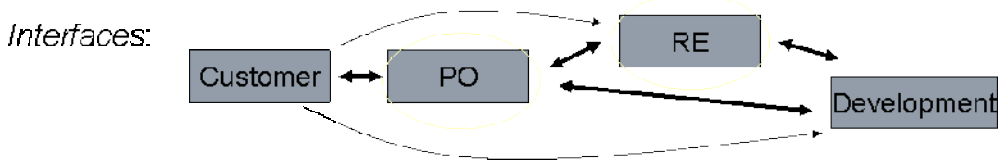


Figure 4.13: Agile RE interfaces

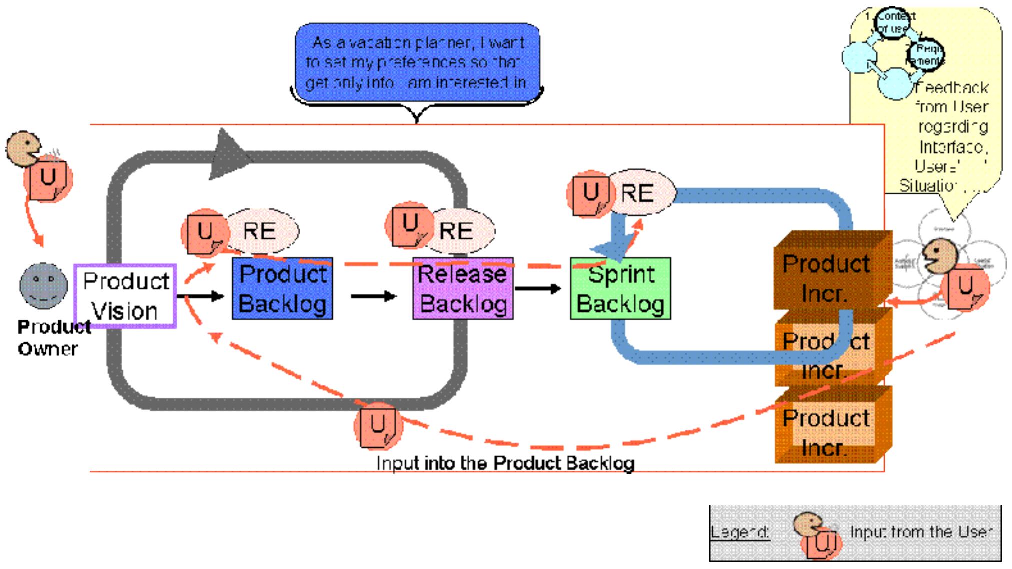

As a vacation planner, I want to set my preferences so that get only info I am interested in
Product Vision
Product Owner
U
RE
Product Backlog
Release Backlog
Sprint Backlog
Product Incr.
Feedback from User regarding Interface, Users' Situation
Input into the Product Backlog
Legend:
Input from the User

Figure 4.14: User Centric RE in Agile's iterative process (enriching the iterative approach of the user-centred RE, described in chapter 4.)

More information regarding Agile RE's' interfaces to customer, stakeholder, or user feedback including Figure 4.15 can be found in [Hoffmann 2011].

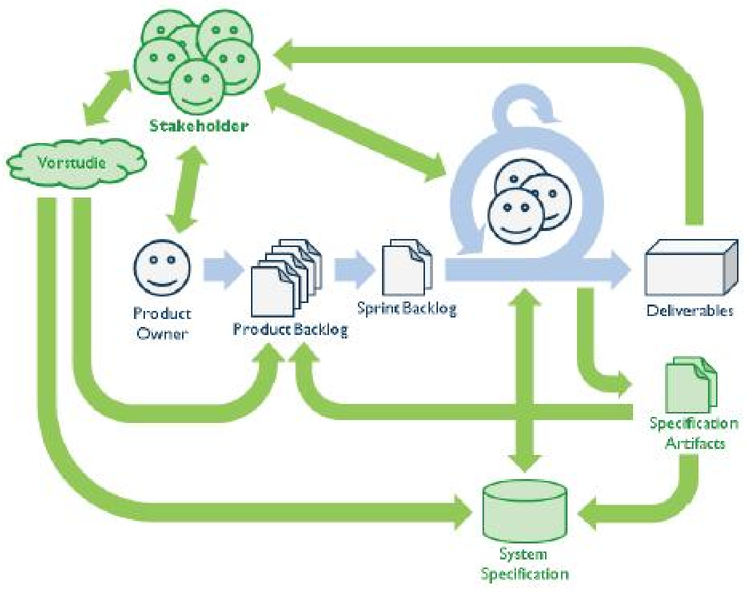


Figure 4.15 A proposal for integration of users from the Sophist group

Let us assume there is a larger organization with some requirements engineering specialists (RE's) supporting the PO and the development team. In order to handle the interfaces from RE's to the development team and PO best, RE processes can be set up in different organizational units.

4.4.2 Requirements Engineering Organization Options

1. RE can be part of a "PO Team" and belong to the Product Owner's organization which could be, e.g., Product Management, or

2. RE can be part of the development team that implements the requirements that come from the PO, and can so be part of the development's organization, e.g., R&D. This option has the advantages that close cooperation and clarification of R&D's needs is easier with the objective to implement user's wishes best. Due to RE's direct dependency on the product backlog and the user stories in the product backlog, the communication and exchange with the PO is mostly good as well.

Note that in all cases there strongly should be one PO as decision point for product vision / product backlog related questions.

Additionally note that PLE is usually a relevant topic in larger organizations.

4.4.3 Product Line Engineering Approach Using SCRUM

The previous sections illustrated different aspects agile RE has to handle (like, e.g., organizational or contextual aspects) - in the case of the development of one single product.

In the case of multi project / more products' development, there are many stakeholders for each project. And, for PLE, all stakeholders' requirements need to be coordinated. So, if PLE wants to be achieved, management decision is necessary,

1. to commit to and support PLE activities, and
2. to establish and empower a PLE Product Owner.

PLE & SCRUM: 1st step

For multi project / more products:

Establish a *PLE Product Owner* who is responsible for the product backlog and release plan/release backlog for PLE. This PLE Product Owner needs to clarify the dependencies, PLE, Re-use, different stakeholders' requirements with the Product Owners of the affected projects / products.

To support the PLE Product Owner best and to manage the requirements from the stakeholders of the affected projects / products appropriately, a "*Stakeholder Analysis Team*" should be established for:

- Coordination
- Which stakeholders are in which project
- Stakeholders' requirements
- Evaluation & Weighting of the Requirements, e.g., in a matrix.
- Dependencies
- Compatibility

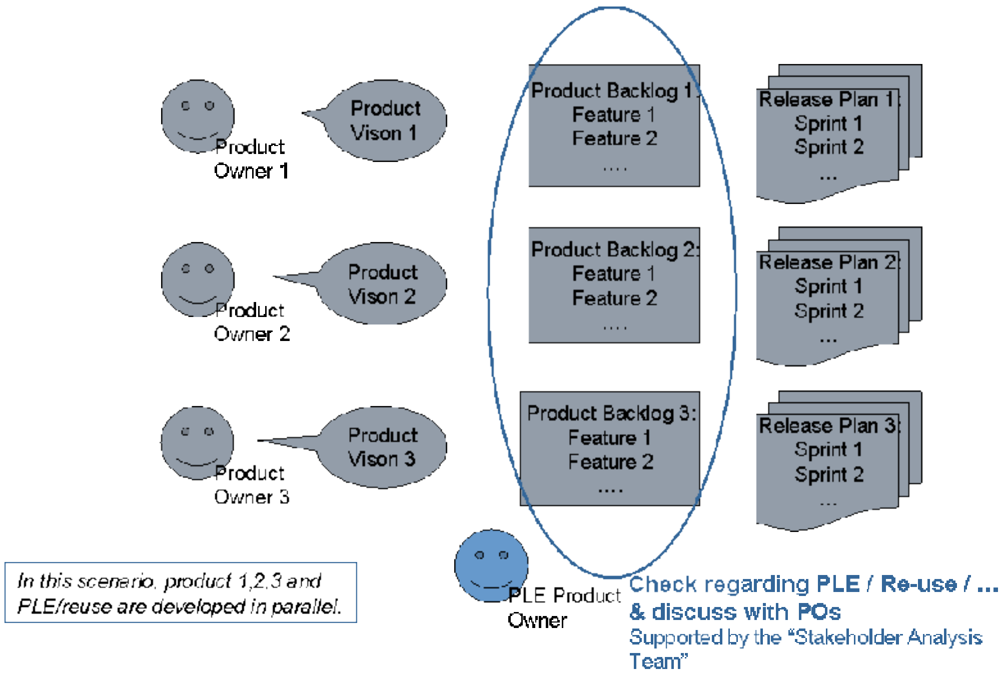

Product Owner 1
Product Vison 1
Product Owner 2
Product Vison 2
Product Owner 3
Product Vison 3
Product Backlog 1: Feature 1 Feature 2 ….
Product Backlog 2: Feature 1 Feature 2 ….
Product Backlog 3: Feature 1 Feature 2 ….
Release Plan 1: Sprint 1 Sprint 2 …
Release Plan 2: Sprint 1 Sprint 2 …
Release Plan 3: Sprint 1 Sprint 2 …
In this scenario, product 1,2,3 and PLE/reuse are developed in parallel.
PLE Product Owner
Check regarding PLE / Re-use / … & discuss with POs
Supported by the "Stakeholder Analysis Team"

Figure 4.16: PLE Product Owner checks Product Backlogs regarding PLE

PLE & SCRUM: 2nd step

The result of the discussion of the affected Product Backlogs with the respective Product Owners, the original Product Backlogs and Release Plans/Release Backlogs might need to be aligned and re-ranked to support the PLE approach best. Also for this, clear management support for PLE is necessary.

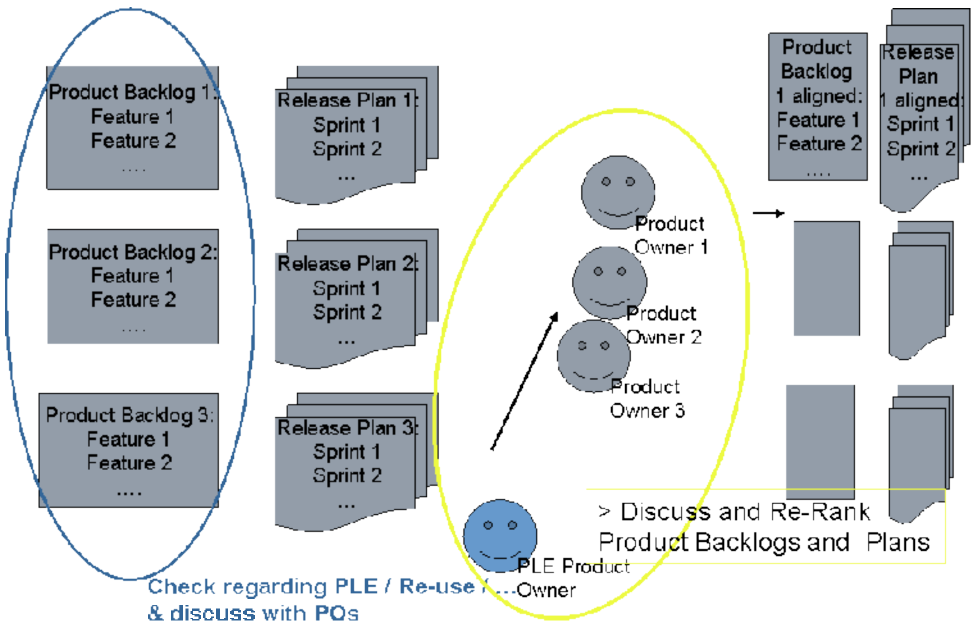


Figure 4.17: PLE Product Owner and POs re-rank product backlogs of the affected products

In further work we will have a closer look the application of SCRUM when introducing a IN-DENICA Virtual Servie Platform

5 ROI Estimation for Service Platforms

‘Platform as a Service’ (PaaS) has a ‘Product Line Engineering’ (PLE) aspect, in that it is provided centrally and used as part of other services at many places, like reuse assets. This comes at the cost of organizing and operating the central installation and the provision of the network infrastructure.

Therefore, when estimating the ‘Return on Invest’ (ROI) of a PaaS approach, similar estimation methods as in PLE can be used. One such method is based on results from the European Union (EU) projects CAFÉ, ESAPS, FAMILIES and is presented in detail in [Pohl2005]. It delivers quantitative ROI analysis results, serving e.g. as a decision base for or against a PLE approach. This analysis method compares the cost of application development in a PLE environment with the cost of developing the same applications in separate uncorrelated development projects. Section 5.1 summarizes this approach. Section 5.2 explains how the formula can be simplified, based on business case constraints of the domain in scope, to become usable. Refer to [Clements2005] and [Bayer1999] for additional information regarding ROI for PLE.

The cost estimation in this method focuses on development costs. This is not sufficient in a PaaS environment, because it does not consider operation, maintenance and infrastructure. Further more the application of PaaS has the potential for additional applications and therefore additional profit that would not be able with distributed standalone applications. Therefore, when estimating the ROI of PaaS, this additional profit should be taken into account too. Chapter 5.3 introduces an enhanced approach that takes account of these additional cost aspects.

If PaaS shall be introduced into an environment that already exists, then a migration strategy is needed and the cost estimation should take this into account. Chapter 5.4 introduces such an approach.

5.1 *Estimating the development ROI for PLE*

Figure 5.1compares a separated product development with a PLE approach. The blue parts are product specific developments; the yellow parts represent the development of reuse assets.

It is assumed that the development of reuse assets happens in a controlled way, and that reuse assets are finally stored in a central location (CM System, File System, etc.), called ‘Core Asset Base’ (CAB) that supports retrieval. Reuse could be (parts of) requirements, plans, specifications, designs, test cases, manuals, subsystems, code, drawings, scripts, or anything else that is reusable.

Figure 5.1 also demonstrates the two main ways of reuse: reuse over time and reuse over space. Reuse over space means that assets are reused across several products or product lines at (roughly) the same time; i.e. in parallel development projects. Reuse over time means that assets are reused in several versions; i.e. in sequential development projects of the same product.

Note that the term ‘Product Line’ has different meanings in different environments. In a PLE environment covers a ‘Product Line’ usually all products or applications that are *based on a common CAB. In a non-PLE environment is a 'Product Line' usually a product with all its versions (and variants) over time. In this chapter (3) the second definition is used, even for the PLE approach.*

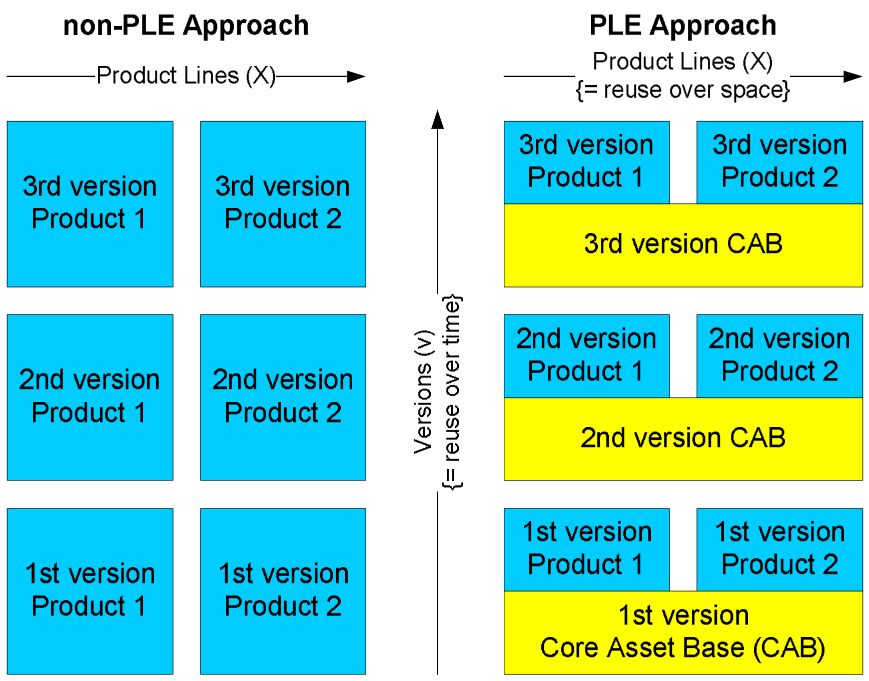


Figure 5.1: Separate product development versus PLE approach

Calculating the cost of (V) versions over (X) product lines is a nested sum over the cost of each product version (C_{prod}):

This formula can be directly used to calculate the cost in a non-PLE environment.

In a PLE approach the cost of a product version is divided into three main parts:

Yellow:	the cost for developing the reuse assets,
Orange:	the cost for using reuse assets (retrieval, adaptation ...) and
Blue:	the development costs for product specific parts.

When migrating from a non-PLE to a PLE environment there is an additional initial cost block (purple) that covers the necessary change costs towards a PLE organisation. Formula 5.2 shows the cost calculation in a PLE environment.

The formula assumes that the number of products or applications using the CAB may vary over time ($X(v)$). Figure 5.2 shows an example for 3 versions and constantly 2 product lines. (see Figure 5.3 for an example with changing number of applications over time).

The formula does not depend on how the CAB development costs ($C_{cab}(v)$) are distributed across the using product lines (e.g. evenly if reuse assets are financed centrally, or depending on grade of usage if financed by licences).

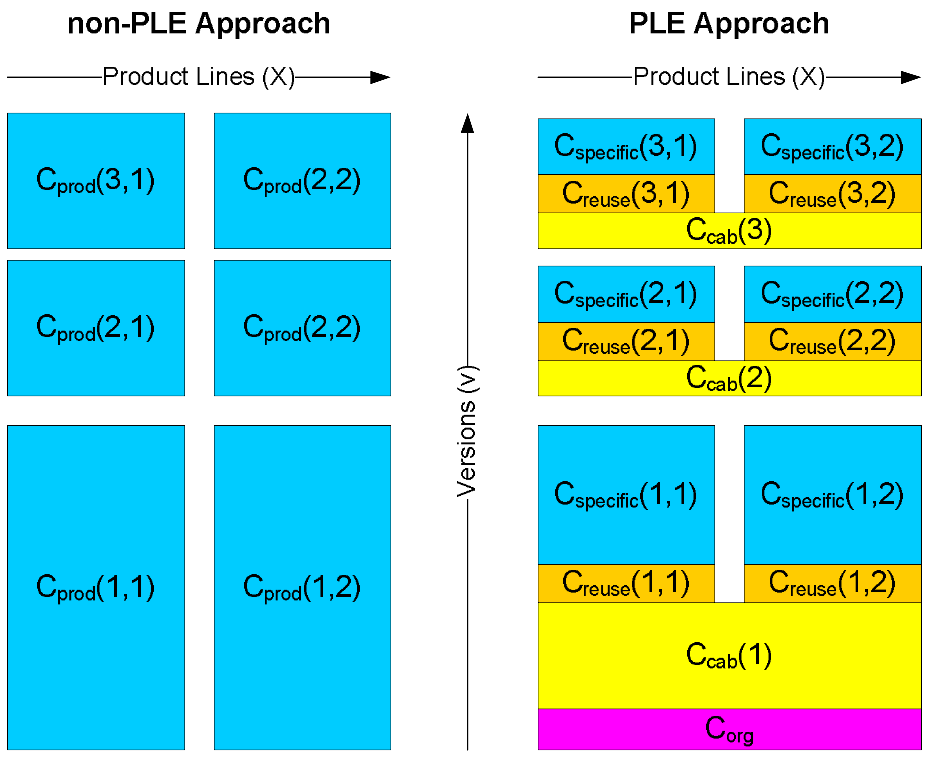


Figure 5.2: Example cost factors for 3 versions of 2 product lines, CAB cost evenly distributed

A 'Return on Invest' (ROI) is calculated by: Gain / Invest * 100%. Formula 5.3 shows this calcultation for the introduction of PLE. Usually the question is how many versions need to be developed to get a positive ROI.

5.2 How to make the PLE cost calculation usable

Formula 5.2 is generic enough to be applied to any PLE development environment. But due to the large number of parameters that need to be estimated it is not really viable. Often the estimation can be simplified by replacing version and product line dependant parameters with (a few) constant parameters. But, such simplifications heavily depend on the business environment.

This sub-chapter describes a simplification that was used in a real project. The goal was to get a rough estimation if it economically makes sense to switch from an existing non-PLE approach to a PLE approach. The result of the calculations was that only for a specific solution the transition would have been recommendable, but not in general.

The business environment had following attributes, which were elaborated together with the domain experts:

- The number of product lines was assumed to be stable over the timeframe in scope
- Reuse assets should have been centrally financed and the costs evenly distributed across the using product lines
- Cost of version development was almost constant for the timeframe in scope, and a known fraction (F_{evo}) of the initial product development; this factor also would have been applicable for the evolution of the core asset base
- Cost for development and test were proportional
- Product cost was dominated by development (incl. test, manuals, …) cost; i.e. there were no planned field maintenance or repair costs after development
- An estimated fraction of assets (F_{reuse}) could have been developed as reuse assets
- The development of reuse assets was estimated to be a factor ($F_{overhead}$) more expansive than a specific asset development (enhanced variability, more documentation, …)

These assumptions translated formula 5.2 to:

In this formula the first version comes at full cost; all other (V-1) versions cost only a fraction (F_{evo}) of the initial development.

C_{org} depended on the number of employees (N_{empl}) and a medium effort per employee (C_{empl}) in relation to working time per year (W_y).

C_{cab} could have been derived from non-PLE development cost, with the assumed factors. It should include the scoping costs ($C_{scoping}$) to identify the reuse assets.

Using reuse assets was not assumed to be free (retrieval, adaptation, …) and the costs was estimated to be a fraction (F_{adapt}) of the specific development cost.

$C_{specific}$ depended on the reuse fraction

The factors N_{empl}, W_y, C_{prod} and F_{adapt} were known by the organization. Experience values from the CAFÉ, ESAPS and FAMILIES projects had been used for the factors C_{empl}, and $C_{scoping}$. Only the factors F_{reuse} and $F_{overhead}$ had to be estimated.

This example shows that the generic formula can be effectively adapted to a certain business case, allowing very efficient PLE migration cost estimations.

Note that this simplification strongly depends on the related domains business case. The simplification may look very different in another domain or depending on the organizations maturity; e.g. there may be non-linear relations or maintenance efforts need to be regarded. Nevertheless cost estimation becomes much easier, if a simplification can be found that eliminates the parameter dependency from ‘V’ and ‘X’. Refer also to [Kreuter2008] for further details of this example.

5.3 Estimating the ROI for a migration to PaaS

It is assumed, that PaaS will lead to additional business, i.e. increased profit "P", due to the enhanced application flexibility. The ROI for a migration to PaaS is calculated in a similar way as for PLE in Formula 5.3.

A possible additional income is completely domain and business dependent.

Regarding lifecycle cost it is a difference if software is operated locally or in a network. Therefore the formula estimating development costs ($C_{development}$) needs to be enhanced by cost term covering usage costs (C_{usage}). C_{cab} represents the ‘platform’ in PaaS.

The development cost is splitted in a reuse and a specific development part, as in Formula 5.2:

The usage costs describe several aspects, the necessary or possible change in IT hardware, the rollout costs of applications that may be locally or network based, and the operation costs of network based applications.

Cost for changes in the IT hardware (C_{it_hw}) includes:

- code overhead in a ‘service environment’ leading to enhanced performance needs
- eliminated redundancies leading to reduced performance needs
- enhanced communication needs

Cost for rollout ($C_{rollout}$) includes:

- trainings of IT service and maintenance
- people change management
- installation and startup
- data migration

Cost for operation ($C_{operation}$) includes:

- rights management
- service distribution

5.4 *Cost estimation in a migration environment*

‘Migration environment’ means that existing applications shall be replaced by PaaS based applications. During the migration phase there will be a fraction of legacy applications besides the PaaS (or CAB) based application for some time, as shown in Figure 5.3. Refer to [Böckle2004], [Bosch2000], [Bosch2002], [Krueger2007] and [Voelter2006] for more information regarding a PLE (migration) strategy.

The migration will also need changes in the development organization, as for PLE, which come for a cost factor C_{org}.

Both parameters need to be included in the formula.

The reorganization costs (C_{org}) include e.g.:

- Reorganization (consider: product management, project management, systems engineering, development of reuse assets, configuration management, IT service and maintenance)
- Motivation and trainings
- Reduced productivity during reorganization time

The cost for maintaining legacy applications (C_{legacy}); include e.g.:

- wrapper code to make it a service in the PaaS world
- bug fixing
- new functionality

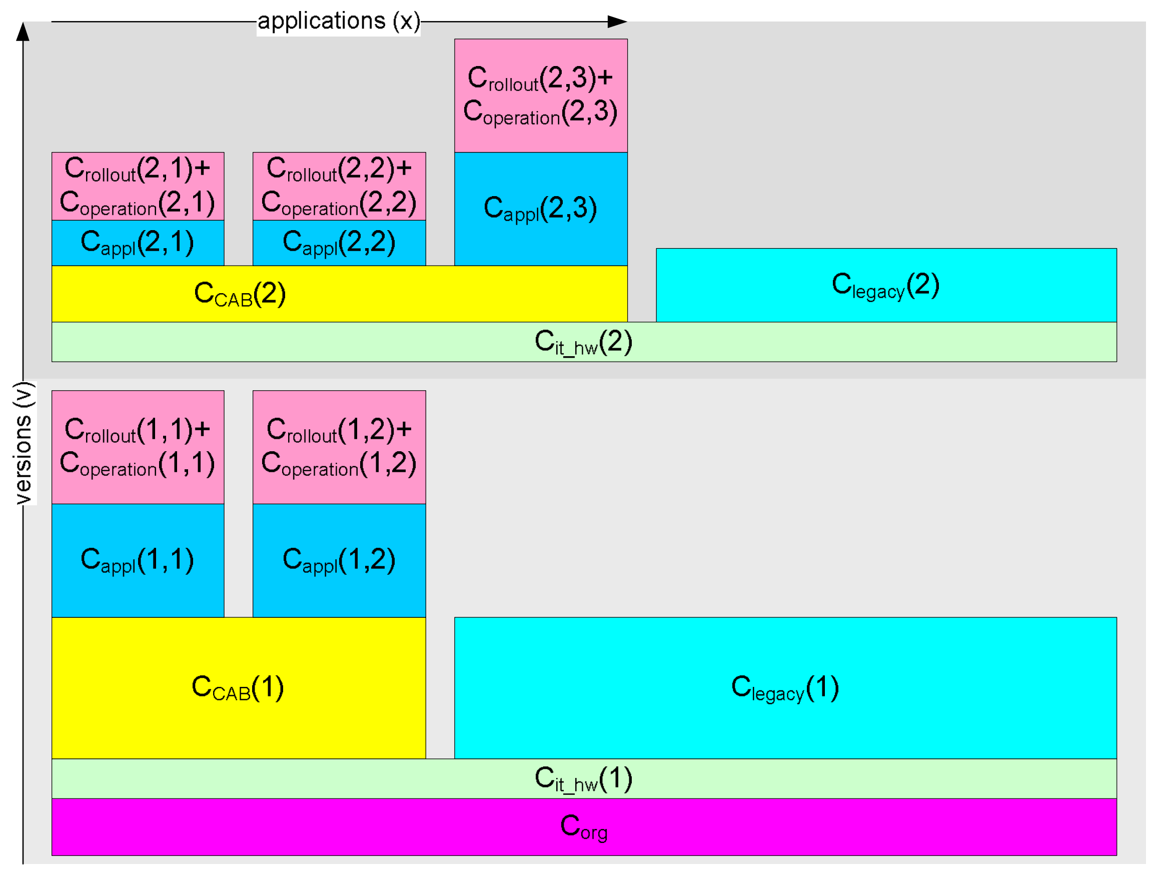


Figure 5.3: Migration of legacy code to a CAB / Platform in PaaS ($C_{appl} = C_{reuse} + C_{specifc}$)

5.5 *Summary and outlook*

The ROI calculation formula developed in the European Union (EU) projects CAFÉ, ESAPS, FAMILIES has been enhanced to cover operation related costs, which is necessary in a PaaS or a Service Oriented Architecture (SOA) environment.

The formula is generic enough to cover a broad spectrum of business cases, even though business specifics may enforce that additional cost parameters need to be included in a similar way. On the other hand is the generality of the formula's parameters an obstacle to really use it as is, because it is hard to estimate all the parameters over typically 5-10 versions of several product lines. Therefore chapter 5.2 gives an example how such a generic formula can be simplified for a dedicated domain and business case, so that it can be efficiently used.

6 References

[Evans 2002] Evans, Meryl K.: Understanding UCD (Interview with Peter Merholz and Nathan Shedroff) http://www.digital-web.com/articles/peter_merholz_and_nathan_shedroff (accessed May 25, 2011.

[Sun 2007] Hongquing Sun: Developing User-centric Software Requirements Specifications. Master Thesis at the Mc Master University, Hamilton, Ontario, 2007

[Kreuter2008] Kreuter et al.: Applying a Cost Model for Product Lines: Experience Report. MESPUL 2008.

[Krueger2007] Krueger: The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines: Proceedings of the 11th International Software Product Line Conference, IEEE Computer Society, 2007

[Voelter2006] Stahl, Voelter: Model-Driven Software Devel-opment Technology, Engineering, Management: Wiley, 2006

[Pohl2005] K. Pohl, G. Böckle, and F. v. d. Linden, Software Product Line Engineering Foundations, Principles, and Techniques. Berlin: Springer, 2005.

[Clements2005] Clements, McGregor, Cohen: The Structured Intuitive Model for Product Line Economics (SIMPLE):, Technical Report, CMU/SEI-2005-TR-003, ESC-TR-2005-003, 2005

[Böckle2004] Böckle et al.: Software Produktlinien: Methoden, Einführung, Praxis, dpunkt-Verlag 2004.

[Bosch2002] Bosch: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, Proceedings of the Second Conference Software Product Line Conference (SPLC2), pp. 257-271, August 2002.

[Bosch2000] Bosch: Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach: Addison Wesley, 2000

[Bayer1999] Bayer, Flege, Knauber, Laqua, Muthig, Schmid, Widen, DeBaud: PuLSE: A Methodology to Develop Software Product Lines: Proceedings of the 1999 symposium on Software reusability, pp. 122 – 131, 1999

[Lauenroth et al. 08] K. Lauenroth, K. Pohl. "Dynamic Consistency Checking of Domain Requirements in Product Line Engineering," Proceedings of the 16th International Requirements Engineering Conference (RE '08)., pp.193-202, 2008.

[Pnuli 77] A. Pnueli. The Temporal Logic of Programs. In 18th Symposium on Foundations of Computer Science (FOCS), pages 46–57, 1977.

[van Lamsweerde 2009] Axel van Lamsweerde. Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley, 2009.

[Hoffmann 2011] Hajo Hoffmann, Requirements Engineering & Scrum, Systemanalyse im agilen Umfeld; in: Marc Sihling, Andreas Rausch, Christian Lange, Marco Kuhrmann (Hrsg.) Software & Systems Engineering Essentials, Proceedings 2011

Appendix

We have successfully tested IRET on Eclipse 3.6 (Helios) and 3.7 (Indigo) on the following operating systems:

- Ubuntu 11.04
- Windows 7

Installation

- Download the IRET zip file from this site:
  https://repository.sse.uni-hildesheim.de/svn/Indenica/deliverables/wp1/d121-release
- Unzip the file
- Open the Eclipse Install dialog (Help → Install New Software...)
- Load the IRET update site from the local folder (see Figure A.1)

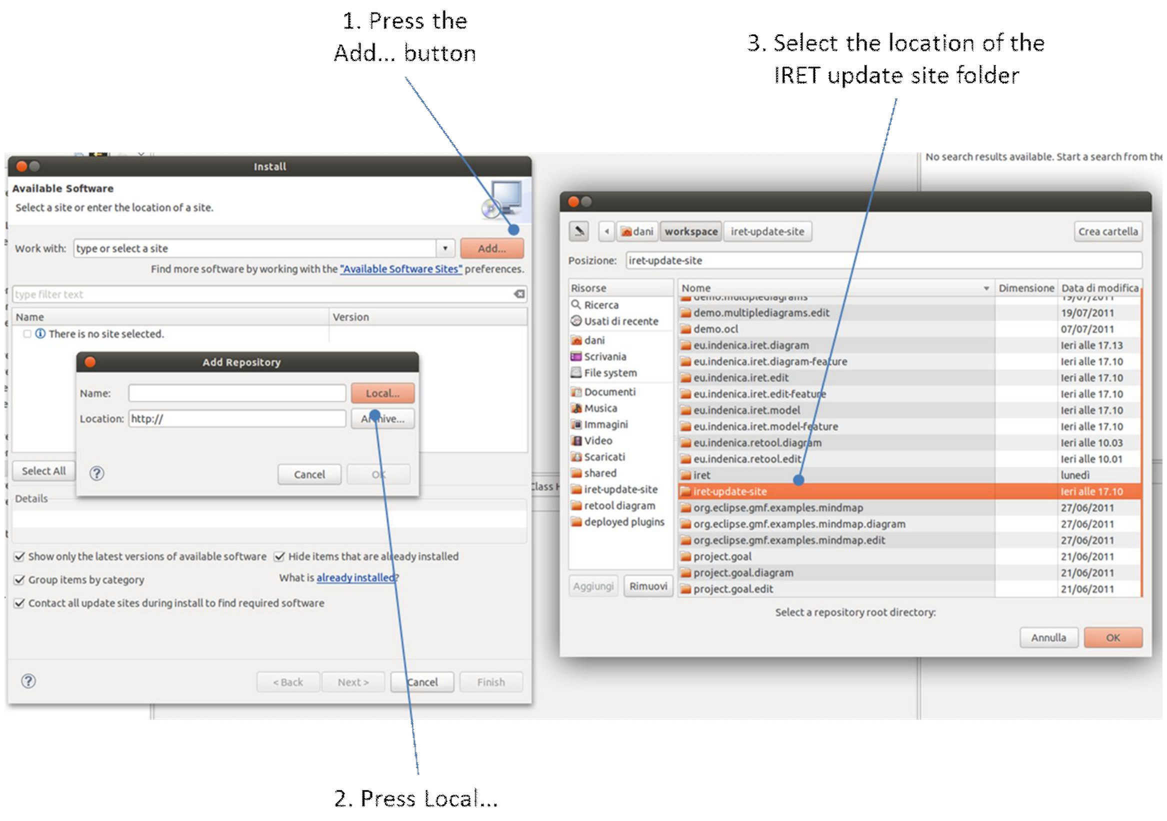

1. Press the Add... button
3. Select the location of the IRET update site folder
2. Press Local...
Install
Available Software
Select a site or enter the location of a site.
Work with: type or select a site
Add...
Add Repository
Name:
Location: http://
Local...
Archive...
Cancel
OK
Back
Next >
Cancel
Finish
iret-update-site
Posizione: iret-update-site
Select a repository root directory:
Annulla
OK

Figure A.1: Load of the IRET update site in Eclipse.

- Check IRENE Toolset to install all the plug-ins composing IRET:
  - IRET Model
  - IRET Edit
  - IRET Diagram
- Press the Next button and follow the wizard procedure: it will install IRET and all the plug-ins it requires to run
- Restart Eclipse