JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 8, November-December 2008

Requirements analysis for large scale systems

Roger Johnson, Dean of the Faculty of Social Sciences - Birkbeck University Of London
George Roussos, Senior Lecture - Birkbeck University Of London
Luca Vetti Tagliati, Technical Lead – Goldman Sachs – UK instead of "Senior Technical Lead - Lehman Brothers (UK)

Abstract

All readers of this paper most likely have knowledge of the software requirements discipline and of the use case notation, however not everyone is aware that with the progress of the development process, requirement models evolve in different ways. In particular, some artefacts (such as the glossary, for example) are simply enhanced, while others (functional requirements, object model, etc), in different stages of the process encapsulate different aspects of the system and satisfy different needs. Therefore, while the former require one to maintain only the most updated version, the latter require the maintenance of different versions (e.g. business functional requirements, domain functional requirements, etc). As expected, this is more evident in the context of large scale systems.

Furthermore, a review of IT literature brings to light some inconsistency in the use of use case notation and, more generally, in requirement models and, even more importantly, to the lack of guidelines related to how to successfully address the requirements analysis for large scale systems.

The objectives of this paper are to clarify the terminology of software requirement models by suggesting a more consistent usage, to describe the different versions of requirement models, and to illustrate how to effectively address requirements analysis for large scale systems.

This paper is supported by a real case study related to a large scale project implemented in a tier 1 financial organisation.

1 REQUIREMENTS MODEL(S)

Background

The majority of currently adopted software development processes, from the least to the most formal ones, give particular importance to the requirement analysis discipline. This is reasonable considering the simple fact that software systems, in the first place, have to be implemented to satisfy specific users' needs. Furthermore, Commercial surveys (for example the Chaos Report [chaos2007]) depict a catastrophic picture of IT project failure: depending on which source is considered, the percentage of project failure varies between 50% to 70%. In addition, these surveys show that a poor requirement analysis is (still) the main reason for project failure.

In the last few decades, a number of methodologies and formalisms have been proposed to support the difficult task of gathering and analysing user requirements. At the moment, a large part of the computer science community agrees in recognising the use case methodology as a *de-facto* standard for capturing and documenting functional requirements. There are even proposals to adopt them within more agile processes like SCRUM ([AGMT2007]).

Software requirements definition and structure

An important prerequisite for this paper is agreeing on what software requirements are and how they are structured. The formal definition of software requirements proposed by the IBM Rational Unified Process (RUP) is: *"A requirement describes a condition or capability to which a system must conform; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document."* [RUP2007]*)*.

A compatible but more elaborate definition is provided by the IEEE [IEEE1998]), where a software requirement is defined as:

A software capability needed by a user to solve a problem or to achieve an objective;

A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document;

The set of all software requirements that forms the basis for subsequent development of the software or software component;

A collective term for the following types of software requirements: functional requirements, performance requirements, external interface requirements, design constraints, and quality attributes.

Software requirement model components

Figure 1 depicts the main components, including the interdependencies, of the software requirements model. The overall software requirements model (depicted by the large package) comprises a number of other artefacts that are modelled as smaller packages (depicted in the figure by a graph of smaller packages linked by dependency relationships). Each of these models, typically, comprises a number of artefacts.

The core of the requirement model is the functional requirement model which describes the intended behaviour of the system. It specifies clearly and in detail each service that the system has to provide. This model includes use case diagrams, part of the

UML notation, with the use case specifications given in terms of a template like the one proposed in figure 2.

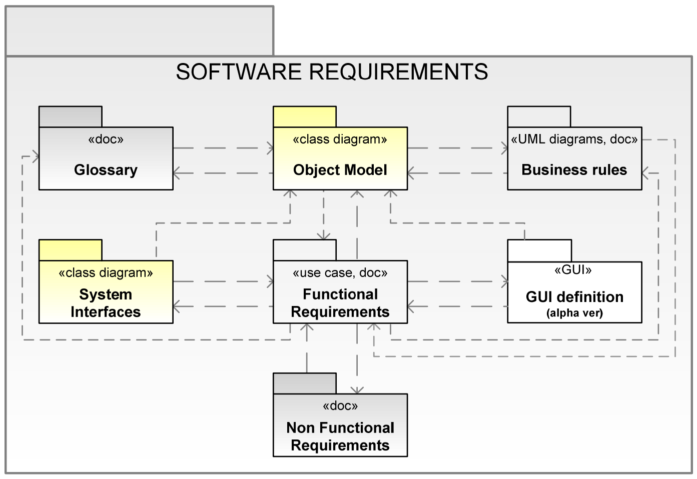

Figure 1. Software requirement model: main components.

The colour notation used in figure 1 indicates the level of formality of the model, in particular:

- yellow packages indicate artefacts that can be entirely modelled using UML;
- white packages indicate packages whose models can be partially designed by UML;
- grey packages include artefacts which cannot be modelled by UML.

USE CASE *<Use case code>*	*<Use case name>*	**Date:**	*<date of the last change >*
		Version:	*<use case version>*
Description:	*<use case brief description>*		
User priority:	*<development priority requested set by users>*		
Performance:	*<requested performance>*		
Primary actor:	*<primary actor name>* *<primary actor interest in this service>*		
Secondary actor:	*<secondary actor name>* *< secondary actor interest in this service>*		
Preconditions:	*<description of the conditions which must be fulfilled before the use case can be executed>*		
Post-condition	**on success:**	*<description of the conditions that describe the state of a system after that this use case is successfully completed>*	
	on failure:	*<description of the conditions that describe the state of a system after that this use case is unsuccessfully completed>*	
Trigger:	*<description of the event that fires the use case>*		
MAIN SCENARIO			
1.	*<actor/ system>*	*<action description>*	
2.	*<actor/ system>*	*<action description>*	
…	*<actor/ system>*	*<action description>*	
n.	System:	Terminates use case successfully.	
ALTERNATIVE SCENARIO: *<condition which leads to this alternative scenario>*			
h.1.	*<actor/ system>*	*<action description>*	
		<action description>	
h.n.	System:	Resumes use case at point x.y.	
EXCEPTION SCENARIO: *<condition which leads to this exception scenario>*			
i.1.	*<actor/ system>*	*<action description>*	
		<action description>	
i.m.	System:	Terminates use case unsuccessfully.	
ANNOTATION			
	<extra information>		

Figure 2. An example of a template for the use case specification as described in the book [LVT2003]

This template allows practitioners to specify the system’s behaviour assuming that everything works correctly (use cases main and alternative scenarios), and the error conditions, including the procedures requested for handling them (exception scenario).

Main (also called success) and alternative scenario definitions allow us to specify and to implement **correct** systems (i.e. systems which effectively deliver the service as requested by the users), while exception scenarios provide us with a fundamental contribution to the delivery of **robust** systems (i.e. systems able to detect anomalies and able to manage them consistently).

Non-functional requirements typically specify properties of the system considered as a whole. This category of requirements includes such aspects as, performance, security policies, system availability, backup and restore strategies, etc. ([IEEE1998]).

The glossary and the Object Model capture the business vocabulary. While the first adopt the natural language, the second is modelled using the UML class diagram notation. As described by I. Jacobson, G. Booch, J. Rumbaugh ([USDP1998]), a domain model captures the most important types of objects in the context of the system. The domain objects represent the ‘things’ that exist or events that transpire in the environment in which the system works”. *Therefore,* “the domain classes come in three typical shapes:

- business objects that represent things that are manipulated in a business, such as orders, accounts and contracts;
- real-word objects and concepts that a system needs to keep track of, such as enemy aircraft, missiles, and trajectory;
- events that will or have transpired, such as aircraft arrival, aircraft departure, and lunch break."

The System Interface model and the GUI model describe the interface used by the system to interact with its actors: other systems and human beings respectively. In this stage interfaces encapsulate only information driven by the business needs and therefore they neglect implementation details.

The common factor of all requirement models and therefore of the overall model, is that they have to specify **everything about what** the system has to do but **nothing about how** it will be implemented.

2 REQUIREMENTS MODELS PROGRESSION

The three versions of the software requirement model components

From the analysis of several industry large scale systems it has been possible to conclude that the different artefacts included in the requirement model can be split into the following groups:

1. artefacts which are **enhanced** with the process and therefore do not require the maintenance of different versions: there is only one version that is enhanced whenever necessary. This group includes: Glossary, Non-Functional Requirements, Business Rules, System interfaces and GUI;
2. artefacts which, in different stages of the process, encapsulate different aspects of the system, satisfying different needs and therefore all versions are important. Models belonging to this group include Functional Requirements and Object Model. The different versions of these models, although linked via a traceable relationships can be considered as different models that have to be maintained during the whole process. For example, a business use case like processing a trade, typically, generates several domain use cases, like "validate a trade", "evaluate a trade risks", "enhance a trade data", "settle a trade", etc.

This paper focuses on the second group.

Three requirement models (figure 3) are identified and presented: **Business**, **Domain** and **System** models, where the latter can be considered a simple evolution of the second (the Domain model is specialised by the System model).

The properties analysed in order to highlight the differences between the three identified models are:

- **boundaries** of the investigated area;
- processes/services of interest;

- level of abstraction.

Only a limited part of the IT literature identifies different requirement/use case models with different properties, purposes, etc. Most of it refers to a sole use case model or requirement document, and others do not even acknowledge the importance of use cases modelling at all. Also, amongst the use case enthusiasts, there is a lack of a common “naming convention” to indicate the different use case models. Therefore the initial aim of this paper is to propose a consistent and rational use case model classification.

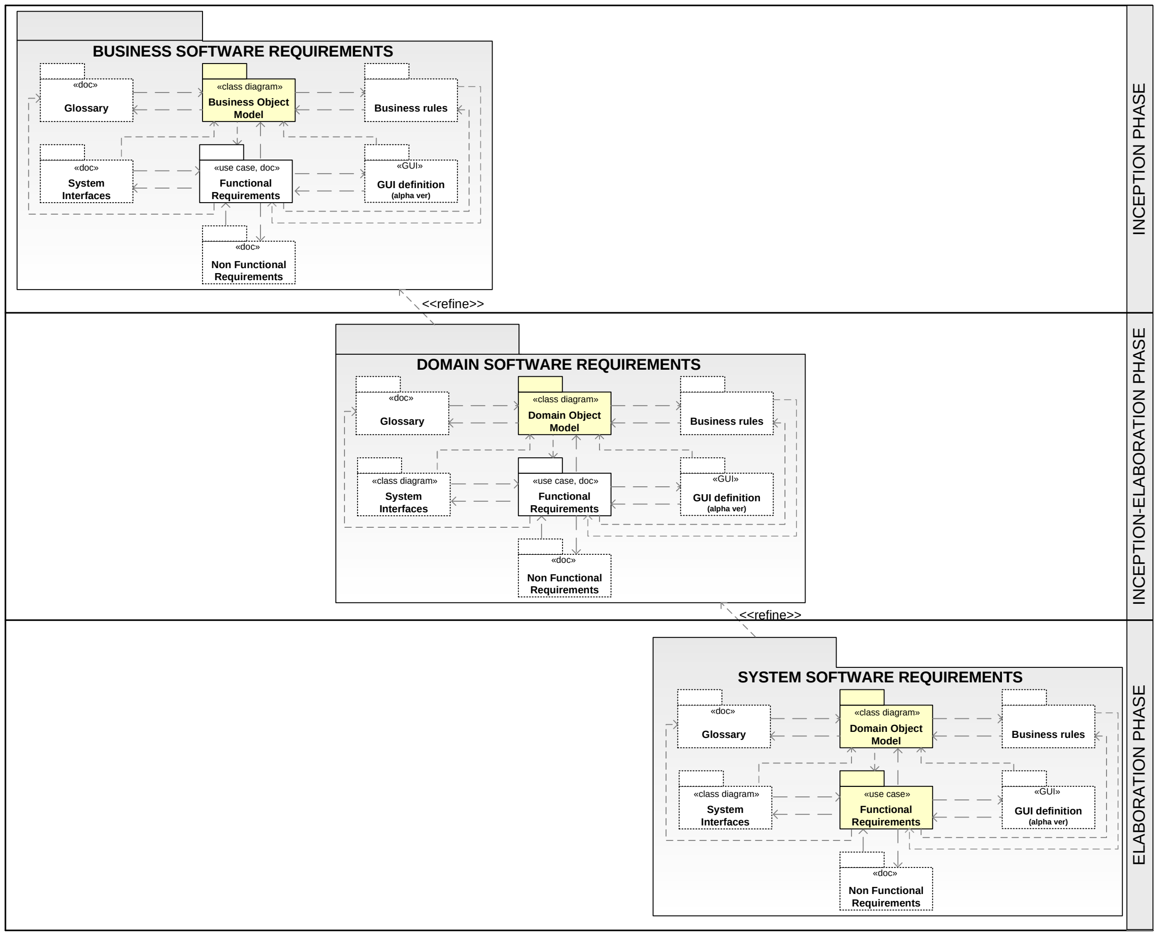
BUSINESS SOFTWARE REQUIREMENTS
«doc» Glossary
«class diagram» Business Object Model
Business rules
«doc» System Interfaces
«use case, doc» Functional Requirements
«GUI» GUI definition (alpha ver)
«doc» Non Functional Requirements
<<refine>>
DOMAIN SOFTWARE REQUIREMENTS
«doc» Glossary
«class diagram» Domain Object Model
Business rules
«class diagram» System Interfaces
«use case, doc» Functional Requirements
«GUI» GUI definition (alpha ver)
«doc» Non Functional Requirements
<<refine>>
SYSTEM SOFTWARE REQUIREMENTS
«doc» Glossary
«class diagram» Domain Object Model
Business rules
«class diagram» System Interfaces
«use case» Functional Requirements
«GUI» GUI definition (alpha ver)
«doc» Non Functional Requirements
INCEPTION PHASE
INCEPTION-ELABORATION PHASE
ELABORATION PHASE

Figure 3. The three requirement models: Business, Domain and System. Artefacts that are simply reviewed and enhanced during the process are represented by a dotted line. Other models which required that different versions are maintained are drawn with a normal line.

Boundaries

Within this context, boundaries indicate the delimitation of the business area considered and analysed by the specific artefact.

The Business model - as the name suggests - focuses on the business area as a whole including the manual process, and not exclusively on the part that the system will automate. Considering a security system for a large organisation, a Business model focuses on the whole area including manual processes that will be only partially automated e.g. new employees reference data gathering, new employees security background checks, etc. In the context of a finance system, a business model focuses on all processes: static data management, pricing, pricing blotter, request for quote, auto-negotiation, trade execution, trade validation and enrichment, settlement, cash flow, and so on.

Progress of the software development process requires the whole business area to be split into smaller domains (where each is typically implemented by a deemed system). This is necessary for a number of reasons, and in particular: to better master the requirements (*"Dividi et Impera"*), to avoid neglecting important requirements and to include feedbacks from the initial architectural blueprints where the system is typically organised in a number of integrated sub-systems. The last point should not be surprising since requirements and the architecture design are mutually dependent. The architecture is designed to accommodate the requirements, but once the design tends to become stable, the requirements must fit into the architecture ([USDP1998], refer to figure 4). Furthermore, the requirements and the architecture must converge. If this does not happen, a number of potential risks are exacerbated. For example several requirements will simply get missed, others could be found to be infeasible for the given architecture, others could have been implemented in more efficient way if only stated in a different way, and so on.

The ideal business model subdivision would generate a perfect partition characterised by each service been allocated to one, and only one, specific domain. Therefore, the same service would not be replicated in various domains i.e. there would be no overlap. This does not mean that once the individual domains (with each domain typically being implemented by a system or a sub-system) have been implemented, the various software components will not interact with each other, but simply that redundancy within the delivered system is minimised.

This desirable scenario is regrettably more the exception than the rule, especially within the context of large scale systems/ large organisation, where, for example, there are a number of systems that cannot be replaced or easily enhanced (e.g. legacy systems and those provided by third parties that cannot be modified). This scenario might lead to the undesirable situation where same services are implemented by different sub-systems and, even worse, implemented using different strategies/algorithms. For example, in large investment banks, it is not infrequent that risk calculation services are delivered by different systems producing inconsistent results or that the front-office receives feeds from different pricing engines. Evidently, this can create severe problems for the business and in implementing SAO systems as well. Therefore, requirements at business level can provide the development team with an important support in designing a better SOA System.

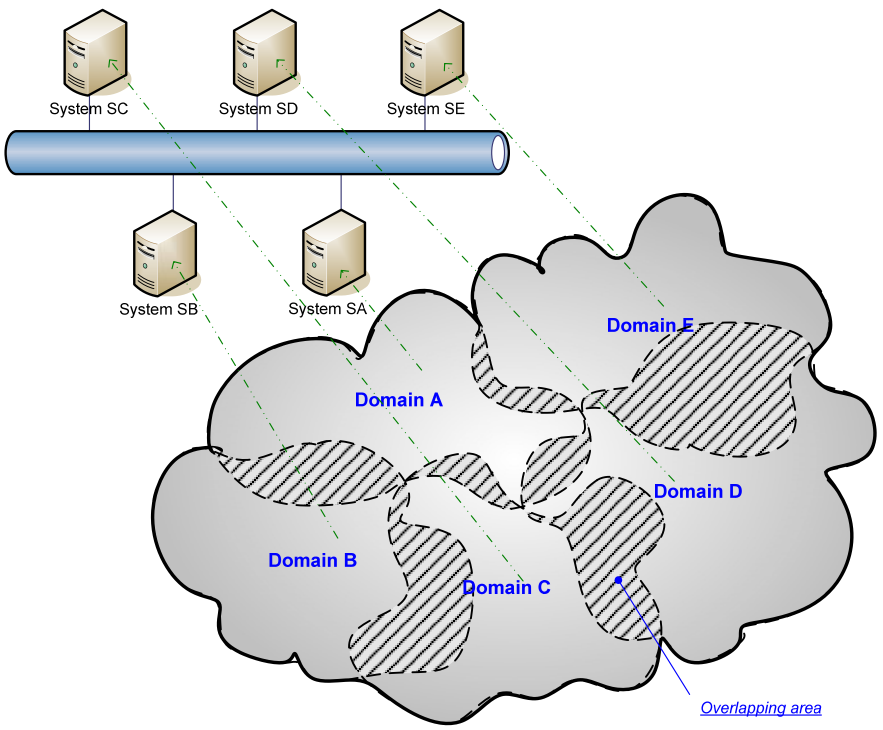

Figure 4. Conceptual partition of a business requirements model into several domains.

Figure 4 depicts a classical business model reparation, where some areas overlap. For example, focusing only on the front-office system of a large investment bank system, the different sub-domains are: static data reference (e.g. counterparty, products), pricing engines, trade execution, trade lifecycle management, trade validation, trade enrichment, trade settlement, cash flow management, positions management, and so on.

This scenario can be applied to several organisations/domains. For example, in the security system, the whole business area can be split into employees/consultant static data (typically managed by HR), personnel profiles management (typically managed by their Manager), daily authentication/authorisation management and data privacy management, and so on.

Business area repartition is not always simple as it may seem. Often, when analysing use case models, one discovers that the system boundaries are ambiguous. For example it is not clear if it is a Business or a Domain model, where the front-office modelling stops and when the back-office begins. Other times, it is quite challenging to establish the boundaries because it is an intricate task deciding which services to include in the system, especially in the first versions.

A consequence of the business repartition is the emergence of a number of new "internal" actors and use cases. In particular, during the specification of the domain use cases for a sub-domain A, the other sub-domains will have to be considered as actors and as result, it will be necessary to consider a number of use cases necessary for the integration of the different areas. For example, system A can be the security system and the other systems can be the HR, the client repository, etc. The required integration tends to generate new use cases. The systems communication functionalities must be implemented and therefore it is better to highlight them as soon as possible instead of dealing with them at the time of implementation. Early discovery of these use cases is important because they allow for a better project estimation, they provide the architect with more information for the system architecture design and also because rarely they are simple data exchange functionalities, more often they include important business rules to be implemented. For example, a billing system requirement typically includes the service of communicating fee reports to the clients. This would be achieved via integration with a report system which, having received billing data would have to analyse the client settings for the communication protocol (such as e-mail and fax), the frequency of the communication, the way to aggregate billing, information and so on. Therefore, it is important that this information management is encapsulated in a specific use case.

Traceable relationship exists between the Business and Domain requirement models and this is particularly important for Change Control Management (CCM) in respect of it being possible to identify and to estimate the impact of a modifications. In particular, if a business requirement changes it is important to be able to asses the ripple effects in all other models.

Considering the use case models, the traceability relationship between business use cases and domain use cases is a "*many to many*" kind:

- some business requirement/use cases do not have a corresponding domain requirement/use case (this is the case of services that have to remain manual);
- others can be decomposed into several use cases in the domain context (for example the same service has to be split in the collaboration of more domains);

the same domain use case can contribute to several business use cases.

Another important difference between Business and Domain models arises from the different levels of abstraction. In particular, Domain models, as expected, must be more detailed.

Domain software requirements model can be further specialised into a System model is obtained by including more architectural feedback (to be precise, typically the process of splitting the Business model into domains is also performed by incorporating an early architecture feedback). The two models are not completely distinct like they are for the Business model. They can be considered to be the two end-points of a logical evolution – by including more and more architectural feedback into the domain use case/requirement model, it evolves into its final version which is the system model. More practically, one can consider these two models like two *base lines* of the user requirements analysis process.

The vast amount of user requirements literature, including this paper, is in agreement on the important guideline that the purpose of user requirements analysis is to specify **what** the system has to do without considering **how** it is implemented. Furthermore, it is clear that specifying the use case/requirements implementation details leads to a number of severe problems. Nevertheless, we state that it is important to include architectural feedback into use case models. Although this might appear contradictory, it is consistent with the guideline. In fact, organising use cases/requirements in accordance with architectural guidelines does not mean that use cases have to specify implementation solutions. It simply requires that requirements correspond more to reality, making it possible to identify, as soon as possible, and to remove requirements that are either not feasible or too difficult or expensive to implement, allowing requirements to be specified in a more convenient way, etc.

This is in agreement with modern software development processes that include an *architecture centric* approach. For example, by establishing architectural guidelines in the early stages of the software development process, it makes it possible to positively influence the requirements analysis. For example it is possible to:

- state that a specific service can be better delivered through a sequence of steps which differs from the proposed sequence;
- state that a different interaction can increase system performance or make the use case easier to implement;
- make the implementation less expensive.

System functional requirements are still described by a technical language closer to the business user world, although IT technical language starts appearing. They do not define implementation solutions but they are specified in a way that is more convenient and consistent with the architecture.

Although it might sound unusual, everyday practice frequently shows that users and business analysts find it easier to specify System functional requirements (write system use cases) than the domain use cases. In the commercial world, it is extremely rare that a new system is developed in an environment where no computer system was ever present. In almost all cases, new projects are related to some kind of system integration or to system reengineering. Therefore, amongst other things, users are somehow aware of factors such as the existing architecture, they know the different systems that interact with each other to deliver services and they use some sort of IT technical language.

When users have a reasonably good vision of the system architecture, it can be worthwhile using this vision – *possibly helping them to use it properly in order to avoid any confusion and/or misunderstanding* - rather than forcing them to use a purer and unnatural language purged of technical terminology.

How to decompose the business model into domain sub-models

The main criteria for dividing up the Business model into Domain models is, as usual, to apply O.O. principles and take into account the architectural guidelines.

Each domain is typically assigned to a specific sub-system designed around high-level cohesive groups of business entities (i.e. data) and a set of functionalities that have to manipulate this data in order to deliver the service requested by users. The task of dividing up the business model is simplified by producing the business/domain object model. By simple analysis it is possible to cluster group of highly cohesive classes which present a low degree of interaction (low level of coupling) with the remaining parts of the system (Figure 5).

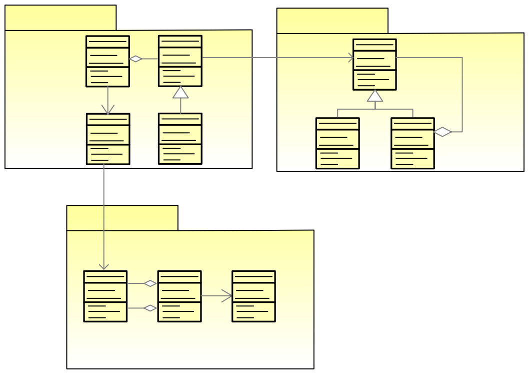
Figure 5. Example of a clustered domain object model

For example, in analysing the domain object model of an investment bank it is possible to cluster classes related to market quotes, static data, trade lifecycle and legal entities, while, in an airline ticketing system, it is possible, for example, to characterise airport related information, customer data, aircraft information, itinerary related data and tickets. It is quite natural to associate each cluster, including the related services, to a specific sub-system/component. However, different clusters of classes are interconnected (they are still part of the same business) and this means that the related sub-systems will almost certainly have to interact with each other in order to deliver services to users.

Once the clustering activity has been carried out the quality of the division should be verified. A valuable test is to allocate each requested service - *use case or steps of use cases, depending on the use case granularity* - to the group of data that it requires. This procedure consists of balancing the domain object model with the use case model ([AUSM200]). This exercise makes it possible to verify basic software engineering laws such as:

- highly cohesive domains (and therefore conceptual subsystems);
- low level of coupling between domains;
- minimisation of subsystem communication;
- logical grouping of services.

When to produce use cases/requirement models

The Business use case model (and more generally the Business requirements model) is one of the very first artefacts that must be produced. In particular, it should be produced during the initial software development phase - the *inception* phase.

Its production is typically a prerequisite to the modelling of the other requirement models (Domain and System). Therefore, there is not much scope for parallel production with the different requirements/use case models. This parallel production is further hampered because it is typically the same team - *the business analyst team* - that produces all three use cases models. However, it is still possible to start producing other artefacts, like the architecture design, which permits greater parallelism. Furthermore, the same requirements model comprises a number of different artefacts that can be developed with a high degree of parallelism.

Once the Business model is delivered, it is possible to start the production of the Domain requirements model which typically takes place during the *elaboration* phase but can also be started in the *inception* phase. The evolution of the Domain use case model into a system use case model results from architectural feedback received from both the *elaboration* and the *construction* phases of the project.

The use case modelling process should terminate with the *construction* phase

Are all models always needed?

This is a rhetorical question and the answer is, naturally, no. However, in order to provide readers with an accurate response, it is important to consider a number of points.

Projects can be split into several categories. For example, there are large-scale projects and medium projects, while others are small. Some projects are more complex than others, many employ *state of the art* technologies, others are more technology-conservative, and so on. What it is important is to note that each project-category has its peculiarities that have to be carefully analysed and addressed before selecting which software development process to apply and how. Indeed, several modern processes are meta-processes and therefore allow a certain degree of customisations. This is necessary in order to allow software development processes to better cope with the specific needs of projects, the development company culture, time-scale, budget and so on. One example of customisation is related to the decision regarding to which requirement models to produce. In theory, it is always beneficial to produce all three models for a number of reasons: for example, a Business requirements model promotes a better understanding of the processes that take place in the business area analyses, it is extremely valuable in implementing proper SOA systems and because it can be used to re-structure the organisation business area, to assess the effectiveness of all business processes and, to train new employees. It is also a great starting point for all future projects and it is the groundwork model to use in building/re-engineering several sub-systems aimed at automating different parts of the business area (domains) and it comprises information, like business data organisation and meaning, which provide software engineers with the base framework for sub-systems integration.

However, it is not always possible to allocate sufficient time and budget, to delay the starting of other development phases while this model is been designed and to allocate all the necessary resources to involve. Therefore, the business model is frequently neglected in favour of the other two. Furthermore, in the majority of commercial projects the sole requirement model produced is the system requirements model (system use cases, system object model, business rules, etc.), typically by a number of iterations of an initial domain model.

However, it is also important to be aware of the fact that whenever artefacts are neglected, automatically certain risks are introduced to the project. For example, not producing the business model can reduce the understanding of the whole business area (often called “the big-picture”), create problems for the future development of other sub-systems (e.g., if there is not a business object model to share) and limit the possibility of integration from the business information point of view, etc.

Nevertheless, "Risk in itself is not bad; risk is essential to progress, and failure is often a key part of learning. But we must learn to balance the possible negative consequences of risk against the potential benefits of its associated opportunity." ([SDR1992]).

Therefore, it is important, throughout the whole development process, to accurately perform “Risk-Management” procedures and to put in place proper mitigation actions… Projects fail because one or more risks have not been addressed properly.

For example, a step aimed at mitigating the risk of a neglected business model, could be to design a number of activity diagrams to represent the end-to-end business processes/services.

On the other hand, a model that cannot be ignored is the domain model. It can be safely stated as a fact that this level of requirements is necessary to implement the system. If this model is not produced, a number of problems can occur. For example, developers will not be able to implement the system without a further undocumented business analysis or without making their own decisions, which turn out to be incompatible with the system architecture and or the business vision. Additionally, it may be impossible to producing a proper integration test plan, to properly estimate the project time and budget, and so on.

3 CASE STUDY

A large investment system

The aim of this section is to present a case study related to a large-scale finance system implemented in a first tier investment bank.

Although the following diagram, for obvious reasons, shows only few business use cases, the real model includes over 250.

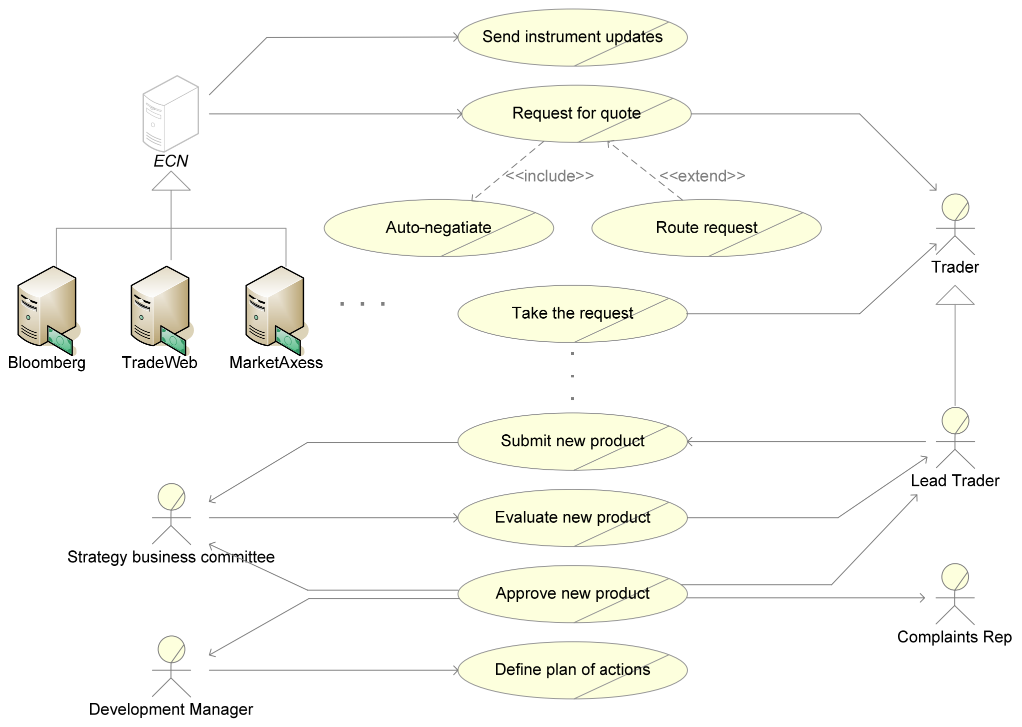

Figure 6. finance system business use cases

The figure 7 shows a simplified version of the business object model clustered into sub-domains. In particular, this version shows some main business "entities" without attributes. Furthermore a number of abstract classes are shown without the corresponding specialisations.

The business object model decomposition provides practitioners with important information for splitting a large scale system into sub-systems. The technique used has been described in the previous paragraphs. The basic principles is to group together highly-cohesive elements, loosely coupled with other groups and, very importantly, manipulated by the same set of service (i.e. use cases).

Each cluster is assigned to a sub-system: it becomes its domain object model. Therefore, that system becomes the owner of the specific object graph. This does not mean that the other systems cannot use the same data but only that the system is the main owner. For example, a number of systems need the reference data to deliver their requirements, but, the owner is the Reference Data Manager system.

Another important consideration is that elements belonging to a cluster are often associated to elements belonging to others. This implies the following:

1. relationships have to be replaced by the migration of IDs;
2. the sub-system that owns the two clusters must be integrated;
3. use cases at domain level will have to include use cases for this integration.

For example, a Contract is between an "own entity" and a "counter-party". Since these classes belong to two different clusters, the corresponding relationships must be broken. In this case it is sufficient to move the own entity and the counter-party natural ID into the class Contract. Therefore the two relationships are replaced by two attributes and integration services.

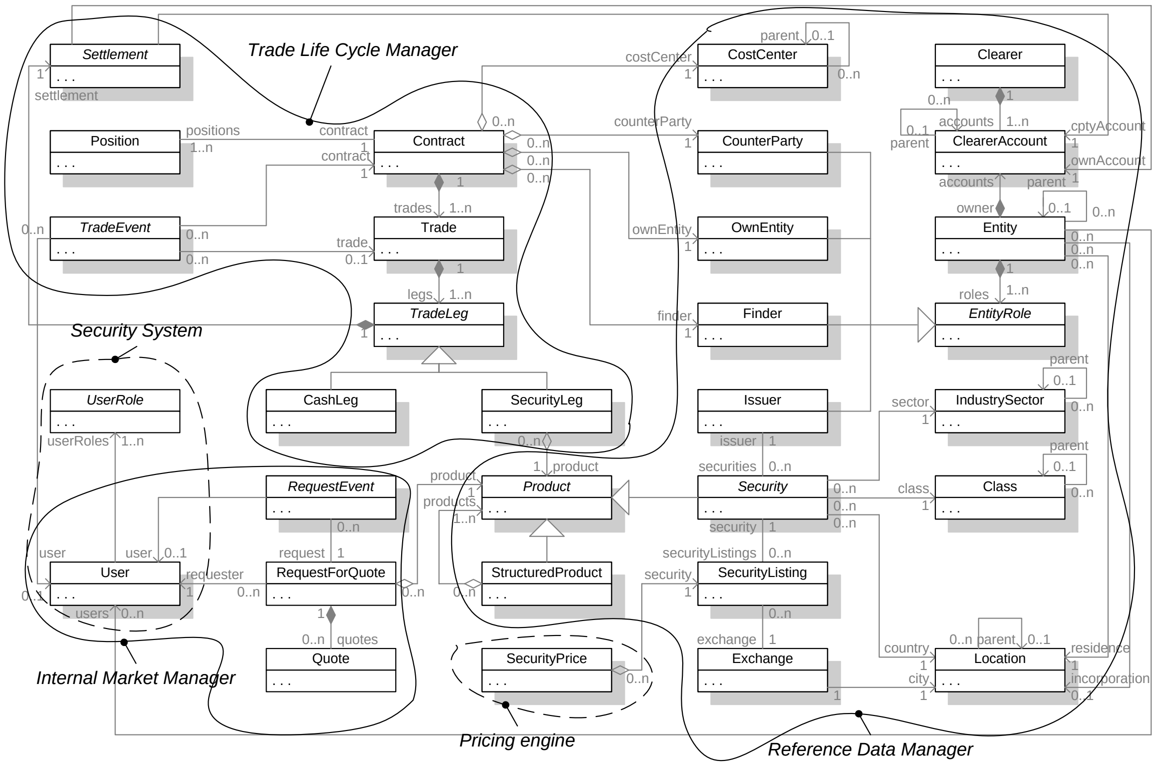

Figure 7. finance system domain object model clustered in domains.

The business object model includes the parts shared by the whole system and therefore it is an extremely valuable artefact in designing system messages and proper SOA architectures.

The following diagram (figure 8) shows the corresponding simplified conceptual architecture. The diagram depicts a system where each sub-system implements only one interface and it depends on only another interface. This is a gross simplification but it is useful for this description. However, it is important to note that among the services that each sub-system implements there are a number related to the need to provide other systems with data belonging to the owned domain object model. For example, the Reference Data System provides other systems with reference data related to instruments, own entities, counter-parties, etc. The Trade Lifecycle Manager provides other systems with data related to contracts, trades, and so on.

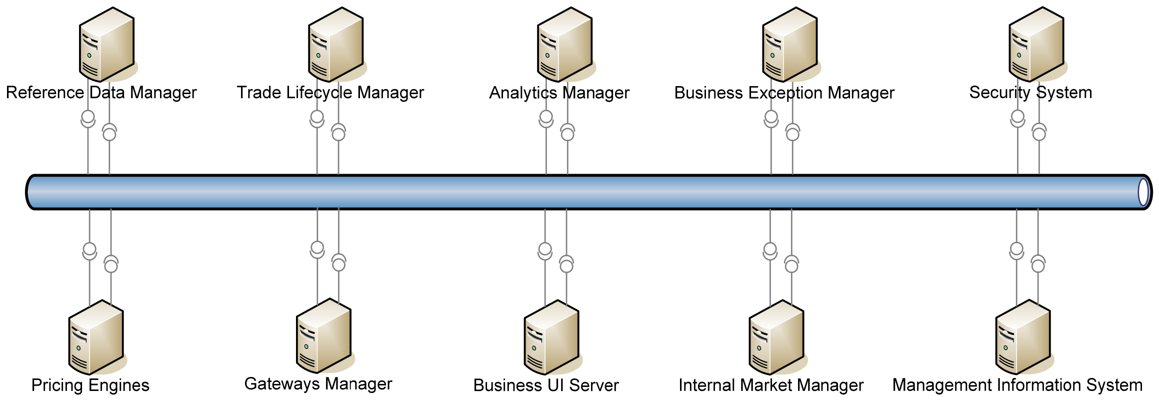

Figure 8. finance system conceptual architecture.

The next step consists of delving into the individual sub-systems, which requires to:

- enhance the assigned graph of elements: the domain object model
- design a new version of the use cases: the domain use cases.

During this process, normally, new use cases new actors appers. Some of these specify the services necessary for the integration and to the fact that when focusing one one sub-system, all other sub-systems are considered actors (refer to figure 9).

As mentioned before, not all business use cases will be implemented. It is normal that a number of services remain manual, at least for the initial phases of the project. For example, in the presented system, it was decided to not implement the process necessary to add new products to the infrastructure as it requires a very complex and fluid workflow. Therefore, some business use cases are not assigned to any domanin.

The following diagram shows some domain use cases related to the Business UI Sever. This system provides UI services mainly to three grops of users: Traders, Sales and Clients. The abstract actor "*Requester*", has been designed in order to emphasise services usable by all actors.

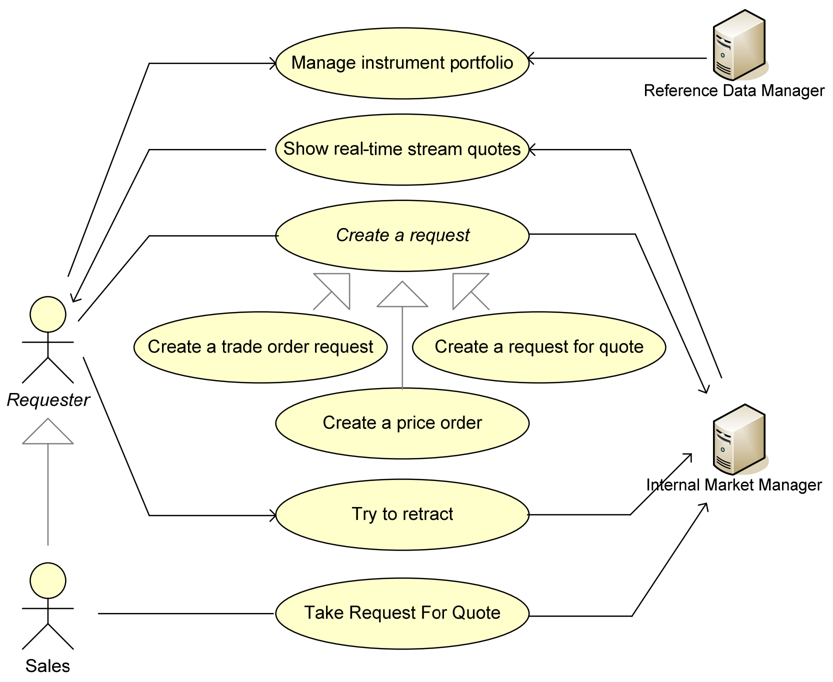

Figure 9. A semplified view of the Business UI Server domain use cases.

4 CONCLUSION

In this paper we have investigated important issues that reside in the vast and complex area of the software requirements analysis and documentation with particular regard to large scale systems. Our first objective was to identify the number of different software requirements/use case model versions that are, explicitly or implicitly, present in a large scale software. In particular, we have identified three use cases models: *Business, Domain and System*. For each we have been discussed guidelines that should clarify the purpose, when, how to produce them, and what are the risk for the project if the specific version is not created.

Due to limited budgets and time constraints, it is not always feasible to produce all these models, although the development of any of these models provides us with significant amounts of valuable information. Therefore, each time we decided not to produce a specific requirement analysis artefact, a project risk is introduced and has to be managed.

The business model is the first that needs to be produced and this is typically done during the inception stage. Its main features are that it is focused on the entire business area (and therefore not limited to the domain that the system has to implement) and that it presents a high level of abstraction. It is not uncommon that commercial projects, due to limited budgets, take the approach of producing directly the domain requirements model. However, this model provides practitioners with important information to design proper SOA system, to design messages, etc.

If the Business use case model is produced, it is purged of manual processes or processes that the system will not implement and it is split into several smaller domains in accordance with O.O. principles.

This exercise is simplified by an analysis of the Domain Object Model with each domain, typically implemented by a specific sub-system. Once the Domain Software Requirements models have been produced, the next stage requires transforming these models into a version of the System. This is accomplished by integrating architecture feedback, which involves updating existing use cases and quite often defining a wholly new set.

By their very nature, user requirements will change but once all the steps outlined above are completed we have the final version of the software requirement models and any changes to requirements will naturally involve the requirements models to be updated.

REFERENCES

[CHAOS2007] The Standish Group International, Inc. - n West Yarmouth, Massachusetts http://www.standishgroup.com/search/search.php

[AGMT2007] Alexandre Lazaretti Zanatta, Paticia Vilain - Agile Methods and Quality Models: Towards and Integration in Requirements Engineering (Seke 2007)

[RUP2007] http://www-306.ibm.com/software/awdtools/rup/

[IEEE1998] IEEE Recommended Practise for Software Requirements Specifications IEEE Std 830-1998

[LVT2003] Luca Vetti Tagliati – "*UML e Ingegneria Del Software. Dalla Teoria Alla Pratica*", Tecniche nuove, 2003 (www.mokabyte.it, www.tecnichenuove.it).

[USDP1998] Ivar Jacobson, Grady Booch, James Rumbaugh - *"The Unified Software development process"* – Addison Wesley

[AUSM2000] Frank Armour, Granville Miller – "Advanced Use Case Modelling. Software Systems", Addison Wesley

[SDR1992] Roger L. Van Scoy – "Software Development Risk: Opportunity, Not Problem". - Software Engineering Institute, CMU/SEI-92-TR-30, ADA 258743, September 1992 – (http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr30.92.pdf).

About the authors

Roger Johnson received a PhD in Computer Science from Birkbeck College. He subsequently worked for a large software house before moving to University of Greenwich and returning to Birkbeck College in 1983. He is a Past President of the British Computer Society and has been active in IFIP and CEPIS. His research interests are in temporal information management, software engineering and computer history.

George Roussos has a BSc in Pure Mathematics (Athens), an MSc in Numerical Analysis and Computing (Manchester) and a PhD in Scientific Computation (Imperial). He was a Marie Curie research fellow, the ISSO for the Hellenic MoD, and the R&D manager for a multinational corporation. His research interests include mobile and ubiquitous computing.

Luca Vetti Tagliati works in the City of London as Technical Lead at Goldman Sachs UK. He is a PhD student at the Birkbeck University of London. His professional career spans more than 15 years, the last 7 have been devoted to the financial domain. He published two books on “UML and the software engineering” and “Java Best Practices”. Over the last few years he had specialised in component based software systems and SOA.