We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

6

A Domain Engineering Process for RFID Systems Development in Supply Chain

Leonardo Barreto Campos[1], Eduardo Santana de Almeida[2], Sérgio Donizetti Zorzo[3] and Silvio Romero de Lemos Meira[4]
[1]*Federal University of Vale do São Francisco (UNIVASF)*
[2]*Recife Center for Advanced Studies and Systems (CESAR)*
[3]*Federal University of São Carlos (UFSCar)*
[4]*Federal University of Pernambuco (UFPE)*
Brazil

1. Introduction

According to the Supply-Chain Council (1997), the supply chain encompasses every effort involved in producing and delivering a final product or service, from the supplier's supplier to the customer's customer. Supply Chain Management (SCM) includes managing supply and demand, sourcing raw materials and parts, manufacturing and assembly, warehousing and inventory tracking, order entry and order management, distribution across all channels, and delivery to the customer. In this context of several sources of information exchanging data dynamically in supply chain, the Radio Frequency Identification (RFID) appears as a technology able to identify objects such as manufactured goods, animals, and people. Thus, the goal of the RFID technology in supply chain management is to guarantee interoperability providing, for example, accurate and real-time information on inventory of the organizations, product recalls and communications among supply chain participants.

On the other hand, the RFID-based systems used in supply chain management were not considered by a specific software development process. In this scenario, a process is important and necessary to define how an organization performs its activities, and how people work and interact in order to achieve their goals. In particular, processes must be defined in order to ensure efficiency, reproducibility and homogeneity (Almeida, 2007). There are several definitions on software process (Osterweil, 1987), (Pressman, 2005), and (Sommerville, 2006). According to Ezran et al. (2002) software processes refer to all the tasks necessary to produce and manage software, whereas reuse processes are the subset of tasks necessary to develop and reuse assets (Ezran et al., 2002). The adoption of either a new, well-defined, managed software process or a customized one is a possible facilitator for success in reuse programs (Morisio et al., 2002). In supply chain domain, many scenarios and processes are repeatable among supply chain participants (sub-domains), for example, inventory management, shipment and delivery of the goods, and localization of a product. In this sense, software reuse – the process of creating software systems from existing software rather than building them from scratch – is a key aspect for improving quality and productivity in the software development.

Source: Supply Chain, The Way to Flat Organisation, Book edited by: Yanfang Huo and Fu Jia,
ISBN 978-953-7619-35-0, pp. 404, December 2008, I-Tech, Vienna, Austria

In the context of software reuse, important research *including company reports* (Bauer, 1993), (Endres, 1993), (Griss, 1994), (Joos, 1994), (Griss, 1995), *informal research* (Frakes & Isoda, 1995), (Frakes & Kang, 2005) and *empirical studies* (Rine, 1997), (Morisio et al., 2002), (Rothenberger et al., 2003) have highlighted the relevance of a reuse process, since the most common way of software reuse involves developing applications reusing pre-defined assets. The software reuse processes literature focuses on two directions: *Domain Engineering* and, currently, *Software Product Lines* (in section 3 a more detailed discussion about it is presented). Thus, motivated by increasing utilization of the RFID technology in supply chain and the lack of specific development processes for the RFID-based systems development in supply chain, this chapter aims at defining a systematic process to perform domain engineering which includes the steps of domain analysis, domain design, and domain implementation.

In the next section we present the parts of the EPCglobal Network. Eight software reuse processes distributed in domain engineering and software product lines are discussed in the followed section. This is followed by an overview of the proposed domain engineering process. The Sections 5, 6 and 7 describe the domain analysis, domain design, and domain implementation steps respectively. Finally, the conclusion summarizes the contributions this work and directions for future works.

2. The EPCglobal network

One critical issue of the new technologies is their standardization. In case of the RFID systems, both EPCglobal and International Standards Organization (ISO) have adopted RFID in their standards. According (Sabbaghi & Valdyanathan, 2008) the most prominent industry standards for RFID are the EPCglobal specifications and standards for supply chain. The EPCglobal Inc is a nonprofit organization that was initiated in 2003 by MIT Auto-ID Center in cooperation with other research universities to establish and support the EPC Network as the global standard for the automatic and accurate identification of any item in supply chain. The EPCglobal is establishing the standards on how information is passed from RFID readers to various applications, as well as from application to application, in the supply chain. These standards are specified in EPCglobal Architecture Framework, or simply EPCglobal Network. Its is a collection of interrelated hardware, software, and data standards, together with core services that are operated by EPCglobal and is delegates, all in service of a common goal of enhancing business flows and computer applications through the use of Electronic Product Codes (Armenio, 2007). It is composed of five components: (i) Electronic Product Code, (ii) Identification System, (iii) EPC Middleware, (iv) Discovery Services, and (v) EPC Information Service.

Firstly, the Electronic Product Code (EPC) is defined as "a naming and identification scheme designed to enable the unique identification of all physical and virtual objects, assemblies and grouping of objects, and non-objects such as service" (Engels, 2003). It is incorporated into a RFID chip and attached to a physical object. An Electronic Product Code is comprised of header and more three distinct numbers: domain manager number, object class number, and serial number. In this way is possible to provide information about product or object such as your category, data and time of manufacture, final destination, etc. The Identification System consists of RFID tags and RFID readers. RFID Tag is an electronic

device composed of microchip and an antenna attached to a substrate, as shown in Figure 1. On the other hand, the RFID readers create a radio frequency field that detects radio waves. When a tag passes through a radio frequency field generated by a compatible reader, the tag reflects back to the reader the identifying information about the object to which it is attached, thus identifying that object.

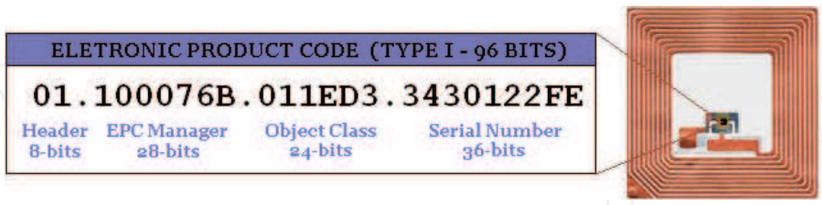

Fig. 1. Electronic Product Code and RFID Tag

Next, the EPC Middleware manages real-time read events and information, provides alerts, and manages the basic read information for communication to EPC Information Services and a company's other existing information systems. The Discovery Services return locations that have some data related to an EPC (EPCglobal, 2005). In general, a Discovery Services may contain pointers to entities other than the entity that originally assigned the EPC code. Hence, Discovery Services are not universally authoritative for any data they may have about an EPC. The important service in Discovery Services is the Object Name Service (ONS) that, given an EPC, can return a list of network accessible service endpoints that pertain to the EPC in question. Finally, the EPC Information Service (EPCIS) provides an uniform programmatic interface to allow various clients to capture, secure, and access EPC-related data and the business transactions which that data is associated (Harrison, 2003). Companies that assign EPC numbers can maintain EPC Information Service servers with item information. Using EPC numbers does not require organizations to share EPC data or use other components of the system.

The EPCglobal Network presented previously contains several aspects that can be considered by Software Reuse. According to (Harrison, 2003), the hardware, software, and Interfaces defined in EPCglobal Network are management by applications with networked databases. In this sense, software reuse can be used in development of applications, reusing assets available in domains of the Supply Chain. For example, the localization of a product in supply chain is divided into six steps: (i) the RFID reader capture the EPC stored on the tag, (ii) EPC Middleware verify and validate the EPC, (iii) EPC Information Service search data related to EPC in local ONS and return the result, (iv) next, the supply chain participant authenticate it in the EPCglobal Network, (v) ONS search data related to EPC in external databases using EPCglobal Network infrastructure, and (vi) return the search results to application as shown in Figure 2.

This scenario and others situation are commons for some supply chain domains. Therefore, the goal of domain engineering process described in this chapter is to identify common and specific features, scenarios, domain-specific software architecture, and so on, for analysts and designers in a domain, as well as simplifying the identification and implementation of the software components.

The next section presents an analysis involving eleven software reuse processes discussing their, fundamentals, concepts, pros and cons that consists a base for the process defined in this chapter.

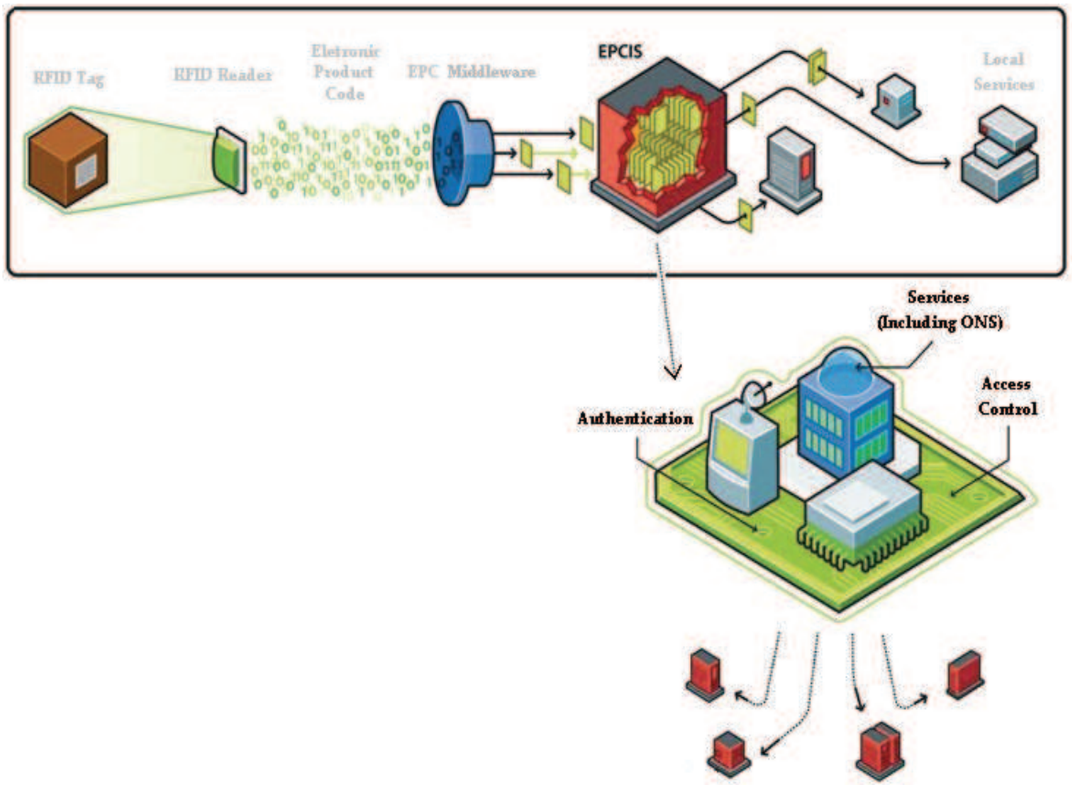

Fig. 2. Scenario of utilization of the EPCglobal Network

3. Software reuse processes

Since the time that software development started to be discussed within the industry, researchers and practitioners have been searching for methods, techniques, and tools that would allow for improvements in costs, time-to-market and quality (Almeida, 2007). The software reuse processes encompass concepts, strategies, techniques, and principles that enable developers to create new systems effectively using previously developed architectures and components. A software reuse process, besides presenting issues related to non-technical aspects (education, culture, organizational aspects, etc), must describe two essential activities: the development *for* reuse (Domain Engineering), which will be discussed next and the development *with* reuse (Application Engineering) which consists in building applications based on the assets produced in Domain Engineering.
In the state-of-art of the software reuse processes presented in (Almeida et al., 2005) and the discussions about it in (Frakes & Kang, 2005) is possible to note that several research studies aimed at finding efficient ways to develop reusable software. These work focus on two directions: Domain Engineering and, currently, Software Product Lines, as can be seen in the next sections.

3.1 Domain engineering

Domain Engineering is defined in (Czarnecki & Eisenecker, 2000) as "*the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets*

when building new systems". In this context, this work analysed four domain engineering processes. The first domain engineering approach is called Draco. This approach is based on transformation technology and was developed by James Neighbors in his Ph.D. Work (Neighbors, 1989). The main idea introduced by Draco is to organize software construction knowledge into a number of related domains. Each Draco domain encapsulates the needs and requirements and different implementations of a collection of similar systems. In this work, is presented also, an initial direction to development software using Domain Engineering. However, his approach is very difficult to apply in the industrial environment due the complexity to perform activities such as writing transformations and using the Draco machine.

Described in the 90's, Feature-Oriented Domain Analysis (FODA) is a domain analysis method developed at the Software Engineering Institute (SEI). The method presented in (Kang et al., 1998) consists of two phases: Context Analysis and Domain Modelling. The major contribution of FODA method is the feature model. An important part of this model is the feature diagram that defines three types of features: mandatory, alternative, and optional features. Next, the Feature-Oriented Reuse Method (FORM) (Kang et al., 2002), seen as an elaboration of the FODA, to present four layers to classify features: capability, operating environment, domain technology, and implementations technique. Moreover, the processes do not include essential domain analysis techniques such as domain scoping.

Other important domain engineering method analysed was the Organization Domain Modeling (ODM) developed by Mark Simos. The ODM process described in (Simos et al., 1996) consists of three main phases: Plan Domain, Model Domain, and Engineer Asset Base. However, the phases do not present specific details on how to perform many of its activities. According to (Czarnecki & Eisenecker, 2000) the ODM provides a "*general high-level guidance in tailoring the method for application within a particular project or organization*".

3.2 Software product lines

A new trend started to be explored in software reuse process is the Software Product Line area. According to (Clements & Northrop, 2001), a software product line is "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*". In this context, we analysed four software product lines processes. Firstly, we present the Family-Oriented Abstraction, Specification and Translation (FAST) process, described in (Weiss & Lai, 1999). The FAST process focuses on pattern for software production processes that strives to resolve the tension between rapid production and careful engineering. The FAST process consists of three well-defined sub-processes: domain qualification, domain engineering, and application engineering. On the other hand, some activities in the process such as in Domain Engineering are not as simple to perform, for example, the specification of an Application Modeling Language.

Presented in (Atkinson, 2000), the Komponentenbasierte Anwendungsentwicklung (KobrA) approach provides a generic assets that can accommodate variants of a product line through framework engineering. The gap in KobrA approach is does not present guidelines to perform systematic tasks such as domain analysis and domain design. An effort to apply the reuse concepts within the embedded systems domain is described in (Winter et al., 2002). The Pervasive Component Systems (PECOS) approach focuses on two issues: how to enable the development of families of PECOS devices? And how pre-fabricated components have

to be coupled? Some gaps was identified in PECOS, for example, in component development, there is not guidance on how the requirements elicitation is performed, and how the components are identified.
Finally, is presented in (Gomaa, 2005) the Product Line UML-Based Software Engineering (PLUS) approach considered the most current process related to product lines. PLUS extends UML by integrating several product line engineering techniques to support UML-based product line engineering. PLUS defines three general steps: requirements, analysis, and design. Modelling in order to provide various modelling techniques and notation for product line requirements engineering activity, use case modelling, and feature modelling. However, in requirements and analysis, activities related to scoping are not considered.

4. The domain engineering process

According to the previous Section, software reuse can be an important way to develop software, offering benefits related to quality, productivity and costs, mainly, using a well-defined reuse process. However, the current software reuse processes present gaps and lack of details in key activities such as, for example, domain engineering. Other requirements like application engineering, metrics, economic aspects, reengineering, adaptation, and quality are important as well, but focus on this chapter is on domain engineering. In this context, this section presents an overview of the proposed domain engineering process, its foundations, and steps.

4.1 Overview of the process

As defined in the previous chapter, domain engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work products), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems (Czarnecki & Eisenecker, 2000, pp. 20). A domain engineering process should define three important steps: Domain Analysis (DA), Domain Design (DD), and Domain Implementation (DI). In general, the main goal of Domain Analysis is domain scoping and defining a set of reusable, configurable requirements for the systems in the domain. Next, Domain Design develops a common architecture for the system in the domain and devising a product plan. Finally, Domain Implementation implements the reusable assets, for example, reusable components, domain-specific languages, generators, and a production process (Czarnecki & Eisenecker, 2000).
Before presenting more details about the proposed domain engineering process, it is important to discuss two basic concepts which will be used next: (i) Domain: encompasses not only the real world knowledge in a given problem area, but also the knowledge about how to build software systems in that area, corresponding to the domain as a set of systems view, (ii) Feature: is widely used in domain analysis to capture the commonalities and variabilities of systems in a domain. In general, there are two definitions of features found in the reuse literature. The first says that an end-user-visible characteristic of a system, the FODA definition. The second definition about feature says that it is a distinguishable characteristic of a concept (e.g. system, component, etc) that is relevant to some stakeholder of the concept, the ODM definition (Simos et al., 1996). In this work, the first definition will be used, since it is the base for the domain analysis area. Other important concepts, more

specific to each step of the process, will be presented later, together with the process detailed description.

4.2 The foundations

A software development process can be understood as the set of activities needed to transform an user's requirements into a software system. The way it is done can change from process to process. For example, processes can be focused on domain engineering, services (Papazoglou & Georgakopoulos, 2003), or Model-Driven Development (MDD) (Schmidt, 2006), or use different process models, however, all kind of process is based on some foundations. In this section, the foundations for the domain engineering process will be presented.

Process Model. A software process model is an abstract representation of a software process (Sommerville, 2006). Each process model represents a process from a particular perspective, and thus provides only partial information about that process. There are different process models published in the software engineering literature, such as Waterfall model, Evolutionary development, Component-Based Software Engineering (CBSE), Incremental delivery, Spiral development, among others (Pressman, 2005). These process models are widely used in current software development practice. The domain engineering process defined in this thesis is based on the spiral process model (Boehm, 1988), however, it presents some characteristics of the CBSE process model, since reusable assets are used to develop applications. The spiral model proposed originally by Boehm (1988), rather than represent the software process as a sequence of activities with some backtracking from one activity to another, consists of a spiral, where each loop represents a phase of the software development process. The Figure 3 shows an overview of the process according to spiral process model. **Domain Driven.** Instead of traditional software development processes as, for example, the RUP, which is use-case driven, the domain engineering process is domain-driven, where the focus is on a set of applications for a particular domain. In this domain, the crucial points are: to identify common and specific features from existing, future, and potential applications; to organize this information in a domain model; next, to design the Domain-Specific Software Architecture (DSSA); and, finally, to implement reusable components for that domain. Even being domain driven, use cases are also used in the process as will be shown in section 6. **Software Architecture.** Software architecture involves the structure and organization by which modern system components and subsystems interact to form systems; and the properties of systems that can be better designed and analysed at system level (Kruchten et al., 2006).

C**omponent-Based Development (CBD).** Component-based Development techniques are important because in domain design, for example, it is interesting to modularize the architecture in well-defined components, which can be easily changed without affecting other parts of the architecture. Moreover, in domain implementation, whose goal is to develop reusable assets, an important way of doing it is through a set of domain-specific components, increasing the reuse potential. **Desig Pattern.** A design pattern is a larger-grained form of reuse than a class because it involves more than one class and the interconnection among objects from different classes. From the perspective of the domain engineering process, design patterns are important because they can be used to encapsulate the variability existing in domain analysis model and perform the mapping for design.

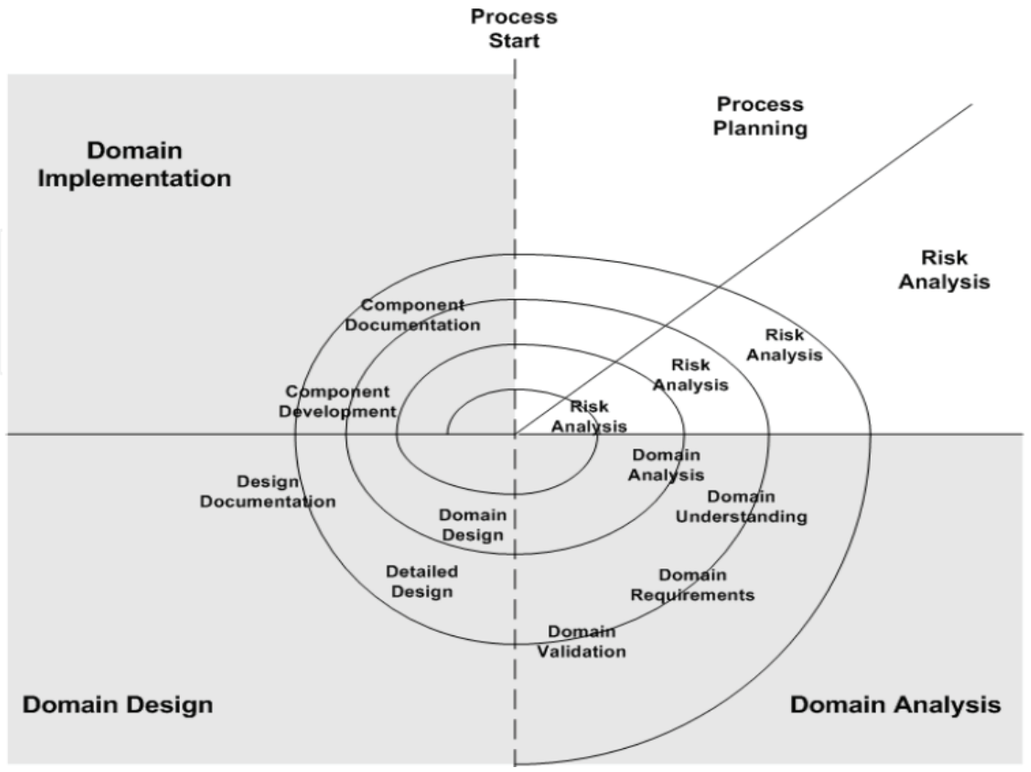

Fig. 3. Process model of the domain engineering process.

4.3 Steps of the domain engineering process

The process for Domain Engineering defined in this chapter is composed of three steps**:** Domain Analysis, Domain Design, and Domain Implementation. Due to the amplitude, each step is respectively divided in activities and sub-activities. The steps are defined to be used in sequence. However, even with less optimal results, they can be used separately. Therefore, in this chapter, each step will be treated as an approach for a different part of the domain engineering life cycle. In this sense, there is an approach for Domain Analysis, described in section 5, an approach for Domain Design, described in section 6, and, finally, an approach for Domain Implementation, described in section 7.

5. Domain analysis step

The Domain Analysis is considered an important phase in reuse-based software development because it is able to identify common and variable features, analyse the domain scope, define the people (stakeholders) who has a defined interest in the result of the project, etc. The first definition for domain analysis was presented by Neighbors as "*the activity of identifying the objects and operations of a class of similar systems in a particular problem domain*" (Neighbors, 1989). For Neighbors, the identification of objects and operations commons in domain minimizes effort in systems development. However, Neighbors not present steps of "*how to perform*" domain analysis.

Based in this gap, works as (Arango, 1989), (Prieto-Diaz, 1990) and (Almeida, 2006) focus on the outcome, not on the process. For Prieto-Diaz (1990), domain analysis is: "*to find ways to*

extract, organize, represent, manipulate and understand reusable information, to formalize the domain analysis process, and to develop technologies and tools to support it". In this sense, this section presents a domain analysis approach for engineering RFID-based systems in supply chain. The goal this approach is identify and modelling commons and variables features presents in each domain supply chain (Campos & Zorzo, 2007). The domain analysis consists of four steps: Planning, Requirements, Domain modelling, and Documentation. The next sub-sections presents each step in details.

5.1 Planning

Firstly, ours goals whit domain analysis are: (i) description of the domain, (ii) identify the stakeholders, and (iii) domain scoping. Therefore, the planning step is based in three sub-activities (*P*):

P1. Domain: encompass to describe which supply chain will be applied the domain analysis (e.g., domain of the healthcare, automotive, food, etc). Next, is necessary to divide the supply chain in domains and describe it into four aspects (*A*): *A1. Activity:* that objective of sub-domain in supply chain? *A2. Input:* from who the sub-domain receives information in supply chain. *A3. Output:* to who the sub-domain send information in supply chain. *A4. Technology:* where and which objective to use RFID systems in sub-domain. Hence, the supply chain will be represented how domains that uses RFID systems in specifics activities, inputs, and outputs. *P2. Stakeholder analysis:* the stakeholders are "*people or someone who has a defined interest in the result of the project*". In this sense, many stakeholders can be identified in development and utilization of RFID-based systems in supply chain. For example, the RFID engineering, person that must be expert with EPCglobal Network, RFID readers, RFID tags and yours variabilities, installation, utilization, etc.

P3. Domain scope: this step consists in identify and discard domains in supply chain out of scope. Domains that do not send or receive data for other sub-domains are eliminated. Next, the domain scope analysis is made in terms of horizontal scope. This type of analysis has the goal answer the questions: How many different systems are in the domain? Finally, the last step, consists of analysing the domain scope in terms of vertical scope. Such, is important answer the questions: Which parts of these systems are in the domain? In this context, vertical domains contain complete systems. An organization which does not have any experience with Domain Engineering should choose a small but important domain. After succeeding with the first domain, the organization should consider adding more and more domains to cover its product lines.

5.2 Requirements

The second step in domain analysis is the requirements elicitation, or simply requirements. The goal is to describe the characteristics of the domain and to understand the users' needs. The requirements identification process includes stakeholders as manager, engineering, and end user identified in planning activity. The Requirements activity is not an easy task, mainly, because can exist some problems in potential since the domain contains several systems, (P*r*) as: *Pr1. Ambiguity*: stakeholders do not know what they really want. *Pr2. Redundancy:* requirements of stakeholders different interpreted of the same form. *Pr3.Conflicting Requirements:* Different stakeholders with conflicting requirements. *Pr4. External Factors:* domain requirements may be influenced by organizational and political factors. *Pr5. Stakeholders Evolution:* news stakeholders may emerge during the analysis process. *Pr6. Requirements Evolution:* change of the requirements during the analysis process.

Our effort to minimize errors is to make the requirements elicitation through features. The feature definition used in this chapter is in concordance with (Kang et al., 1998): "*an end-user-visible characteristic of a system*". After defining the form to extract the domain requirements, is necessary to define as to extract them. In this task, the approach uses the concept of the scenarios. The scenarios are descriptions of how a system is used in practice. Thus, the steps (*S*) for requirements elicitation from scenarios are: *S1. Initial stage*: systems stage at the beginning of the scenario. *S2: Events*: normal flow of events in the scenario. *S3. Alternative Events*: eventual events out the normal flow that can cause error. *S4. Finish stage*: systems stage on completion of the scenarios. *S5. Stakeholders*: to list the stakeholders that had participated in scenarios.

5.3 Domain modelling

The Domain Modelling is the third step in domain analysis. Your goal is identifying and modelling commons and variables requirements in domains. In RFID-based systems, the features will be based on the EPCglobal Network and the following sub-activities (*M*): *M1. Commonality analysis:* consist in identify which features are commons to all applications of the domain. There are different ways to identify common requirements. This approach uses a based-priority sub-domain-requirements matrix shown in Table 1. The idea is select requirements by priority for all stakeholders.

	Dom. 1	**Dom. 2**	**Dom. 3**	**. . .**	**Dom. n**
Req. 1	Pr2	Pr1	Pr2	-	-
Req. 2	X	Pr2	Pr3	-	-
Req. 3	Pr1	Pr1	Pr1	-	-
. . .	-	-	-	-	-
Req. n	-	-	-	-	-

Table 1. Structure of Based-Priority Domain-Requirements Matrix

The left column of the matrix lists the requirements of the considered sub-domains. The sub-domains themselves are listed in the top row. In the body of the matrix it is filled by priority of the requirements. The priorities (*Pr*) are classified as follows: *Pr1. High:* the requirement '*Pr1*' is mandatory for all sub-domains and is thus a candidate to be defined as a common domain requirement. *Pr2. Medium:* the requirements that assists high-priority requirements to keep the functionality of the systems. *Pr3. Low*: low-priority requirements to systems. After filling of the matrix, the domain analyst must define ideal priority for commons requirements.

The second sub-activity is *M2. Variability analysis:* this activity consists in identifying which features are variable to applications of the domain. According to (Svahnberg et al., 2001) in situations where a lot of effort has been made to preserve variability until very late in the development process, the systems provides greater reusability and flexibility. Finally, we have the sub-activity *M3. Domains modelling:* here, the commonalities and variabilities are modelled. The model may be applicable at a high level to a number of applications. In this approach the features may be *mandatory, optional, or features* or *alternative* as shown Figure 4 (Czarnecki & Eisenecker, 2000):

According to Czarnecki and Eisenecker (2000) a *mandatory feature* node is pointed to by a simple edge ending with a filled circle. An *optional feature* may be included in the description

of a concept instance if and only its parent is included in the description. A concept may have one or more sets of direct *alternative features*. Finally, a concept may have one or more sets of direct *or-features*. However, if the parent of a set of or-feature is included in the description of a concept instance, then any non-empty subset from the set of or-features is included in the description; otherwise, none are included.

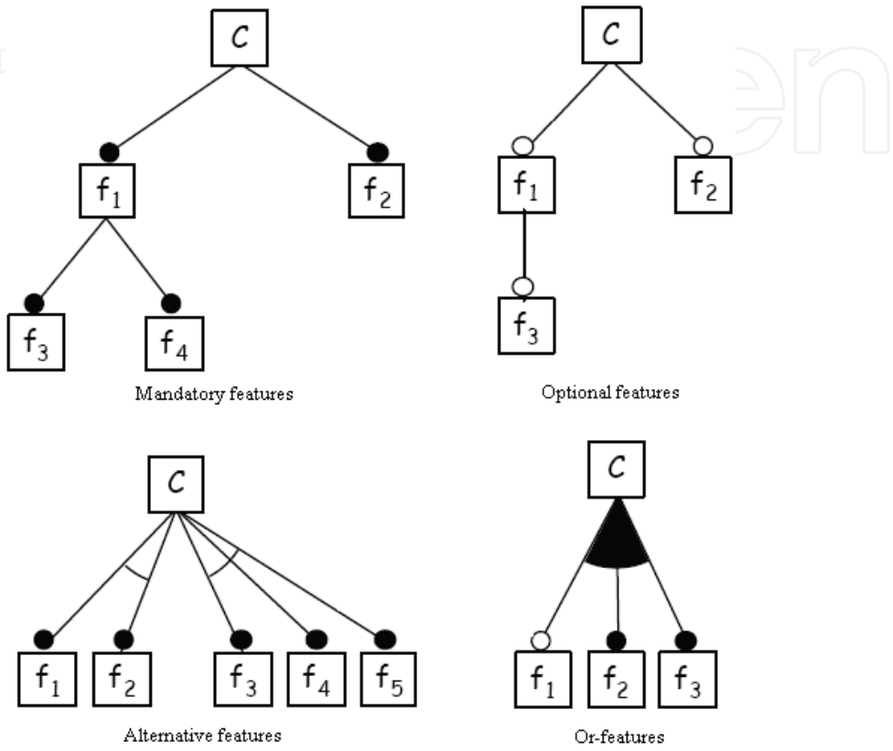

Fig. 4. Features types. Adapted (Czarnecki & Eisenecker, 2000)

Documentation. In this activity the requirements, identified in form of features, will be documented. According to (Czarnecki & Eisenecker, 2000) the template used for document features contain the fields:

6. The domain design step

The second phase of the domain engineering process defined in this chapter is the Domain Design. The key goal this phase is to produce the domain-specific or reference architecture, defining its main software elements and their interconnections in concordance with (Bosch, 2000). The concept of software architecture as a distinct discipline started to emerge in 1990, and in 1995 (Shaw & Garlan, 1996), the field had a strong grow with contributions from industry and academia, such as methods (Kazman et al., 2005) for software architecture. Our domain design approach use the following concept: "*A software architecture is a description of the subsystems and components of a software system and the relations between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artefact. It is the result of the software development activity*", presented by (Clements et al., 2004).

Feature Name:
Semantic Description
Each feature should have at least description describing its semantics
Rationale
A feature should have a note explaining why the feature is included in the model
Stakeholders and client programs
Each feature should be annotated with stakeholders (e.g., users, customers, developers, managers) who are interested in the feature and the client programs that need this feature
Exemplar applications
If possible, the documentation should describe features with known applications implementing them
Constraints
Constraints are had dependencies between variable feature. Two important kinds of constraints are mutual-exclusion constrains and required constrains
Open/closed attribute
Variation points should be market as open if new direct variable sub-feature (or features) are expected. On the other hand, marking a variation point as closed indicates that no other direct variable sub-feature (or feature) are expected
Priorities
Priorities may be assigned to features in order to record their relevance to the process

The main way of reusing a software architecture is to design a Domain-Specific Software Architecture[1] (DSSA) (Tracz, 1995) or Product-Line Architecture[2] (Dikel et al., 1997). The difference between software architecture in general and a DSSA is that a DSSA is used by all applications in the domain. In this sense, a DSSA for RFID-based Systems in Supply Chain is defined in the domain design phase and your goal is develop an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure effective for building successful applications (Tracz, 2005). The next sections present the activities of the domain design: (i) Mapping, (ii) Components Design, (iii) Architecture Views, (iv) and, Architecture Documentation.

6.1 Mapping

The first activity in domain design is the mapping from requirements to reference architecture. An important issue considered in this activity is the variability. According to

1 Term used by the reuse community and adopted in this thesis.
2 Term used by the software product lines community. However, both present the same idea

(Svahnberg et al., 2001), variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. In domains supply chains there are many variabilities, both in use of the RFID technology and in supply chain organization. Therefore, the requirements mapping must keep the variability in order to repeat the process for many different domains and offer a reference for it.

Other issue that the domain designer should consider is with components specifications. Some decisions (e.g. algorithms used in component development, objects and types of the component interfaces) can to restrict the component reuse. When these decisions conflict with specific requirements, the components reuse is limited or the system will be inefficient. An efficient way to minimize or eliminate these conflicts is using Design Pattern. According to Christopher Alexander (Alexander et al., 1977) "*each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million tomes over, without ever doing it the same way twice*". This way, the pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. In the software world, design pattern were popularized by the gang of four (Gamma et al., 1995). According to Gamma and his colleagues, design patterns describe a recurring design problem to be solved, a solution to the problem, and the context in which that solution works. From the perspective of the domain engineering process, design pattern are important because they can be used to encapsulate the variability existing in domain analysis model and perform the mapping for design. In general, a pattern has four essential elements: the pattern name, the problem, the solution, and the consequences.

In the approach presented in this chapter, Design Patterns are used, but together with useful guidelines that determine how and when patterns can be used to represent the different kinds of variability that can exist in a DSSA for RFID-based Systems in Supply Chain. In to order to design the variability of each module, we consider that it should be traceable from domain analysis assets (features) to architecture, according to alternative, or and optional features (Lee & Kang, 2004).

6.1.1 Alternative features

Alternative features indicate a set of features, from which only one must be present in an application. Thus, the following set of patterns can be used (Gamma et al., 1995): **Abstract Factory.** The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Specifying a class name when the domain designer creates an object commits you to a particular implementation instead of a particular interface. In this way, this pattern can be used to create objects indirectly and assure that only one feature can be present in the application. In RFID-based systems in supply chain, there are several readers and simultaneous readings. Thus, the EPC Middleware must select only one EPC in case of various requisitions and to discard unnecessary information of the data bases as shows Figure 5.

Chain of Responsibility. This pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Objects in a chain of responsibility must have a common type, but usually they do not share a common implementation. In this sense, the same requisitions realized in distinct domains can be

resolved by different objects. For example, the Discovery Service can want to be able to query the data in local ONS or in external databases. **Factory Method.** Defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern is similar to the abstract factory and can be used also for alternative features. Finally, **Observer** defines an one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Using this pattern, features can be added to the application as a plug-in, after the deployment. In supply chains, the systems must be flexible the various patterns RFID tags used in identifying of the products or items.

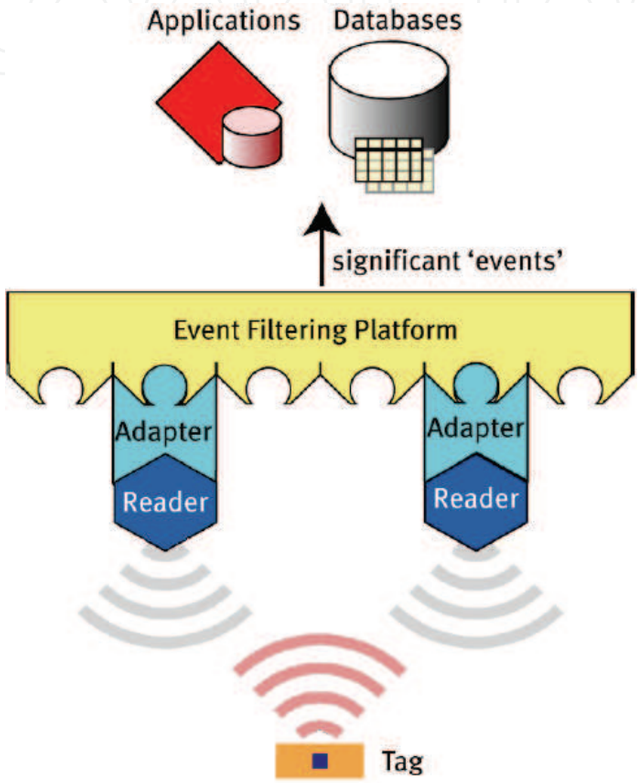

Fig. 5. Simultaneous readings. Adapted (Harrison, 2003)

6.1.2 Optional features

Optional features are features that may or may not be present in an application. For this type of feature, three patterns can be used (Gamma et al., 1995):

Decorator. Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classes for extending functionality. The decorator pattern can be used for optional features, mainly those that are additional features. Thus, if a feature is present, the ConcreteDecorator is responsible to manage and call the execution. **Prototype.** Specifies the kinds of object to create using a prototypical instance, and create new objects by copying this prototype. The prototype pattern specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. In this pattern, the prototype specifies how the interaction wit the feature should be, by defining a concrete prototype for each feature. When the EPC Information Service request

data of any EPC to Object Naming Service and it does not provide information, data obtained of the external databases are copied in local server. **Observer.** This pattern can be used in the same way as in alternative features.

6.1.3 Or features

Or features represent a set of features, from which at least one must be present in an application. For this type of feature, three patterns can be used (Gamma et al., 1995):
Bridge. Decouples an abstraction from its implementation so that the two can vary independently. This pattern is appropriated where exist dependence on object representations or implementations, and dependence on hardware and software platform. **Builder.** The pattern separates the construction of a complex object from its representation so that same construction process can create different representations. This pattern can be used to build composed features. Thus, for the remainder of the architecture, only the Director is available, being responsible to decide which features will be in the application and which will not.. For example, in the transport of pallets, the application decides what transport unit will be utilized (truck, ship, aeroplane, etc), and creates the object automatically considering its characteristics (size, weight, etc). **Singleton.** Ensures a class only has one instance, and provide a global point of access to it. This pattern also is strongly recommended to or features that interact with mandatory features.

6.2 Component design

In this activity, the goal is to create an initial set of interfaces and component specifications. This activity is composed of two steps: Identify Interfaces, and Component Specification. Firstly, is important understand the concept of interfaces. For (Szyperski et al., 2002) define interface as "*a set of operations, with each operation defining some services or function that the component will perform for the client*". In concordance with (Cheesman & Daniels, 2000) our work considers two types of interfaces: system and business. The business interfaces are abstractions of the information that must be managed by components. Our process for identifying them is the following: to analyse the feature model to identify classes (for each module and component); to represent the classes based on features with attributes and multiplicity; and refine the business rules using formal language. The system interfaces and their operations emerge from a consideration of the feature model and mainly of the use case model. This interface is focused on, and derived from, system interactions. Thus, in order to identify system interfaces for the components, the domain architect uses the following approach: for each use case, he considers whether or not there are system responsibilities that must be modelled. If so, they are represented as one or more operations of the interfaces (just signatures). This gives an initial set of interfaces and operations.
After identifying the interfaces, additional information for specifying components are necessary as, for example, interdependency of the components, and interfaces. The steps presented in this chapter to identifying the interfaces are in concordance with (Cheesman & Daniels, 2000). Firstly, for every component that is specified, the domain architect defines which interfaces its realizations must support (provided and required interfaces). Next, the restrictions of interaction between components must be specified. Unlike the traditional interactions in implementation level, interactions of components define restrictions on the specification level.

6.3 Architecture views

A good way of mapping requirements to implementation is across of the architecture views. The view must be defined as a representation of a set of system elements and the relations associated with them. A view constrains the types of elements, relations and properties that are represented in that view. In this work the four views considered are: (i) module view, (ii) process view, (iii) deployment view, and (iv) data view.

The module view shows structure of the system in terms of units of implementation (e.g. component, class, interfaces and their relations). It is essentials because represents the blueprints for software engineering. Despite of the EPCglobal Network to propose one architecture reference, it is can contain different modules in domains. The module view defines three types de relations in concordance with UML relations between modules (Jacobson et al., 1999): *is part of*, *depends*, *is a* as shown the Figure 6. The first relation "is part of" is used when a package contain sub-packages and class. The second relation "depends of" show the dependences between modules, for example, if the EPCIS need to update your data bases are necessary to authenticate in EPCglobal Network. Finally, the relation "is a" have as goal to represent specialization or generalization among modules, or interface realization. The notation more appropriate to represent module view is although UML diagrams as: package, components, class, and objects diagrams.

The second architecture view defined in this chapter is the runtime view. It shows the systems in execution, your properties, performance, and help to analyse some features in runtime. The best representation for this view is using the UML diagrams following: Interaction, Timing, State Machine, Activity, Communication, and Sequence diagrams. Together with the activity diagram, the state machine diagram to offer more features to describe the process exists in RFID-based systems of the supply chain. These diagrams depict behavioural features of the system or business process.

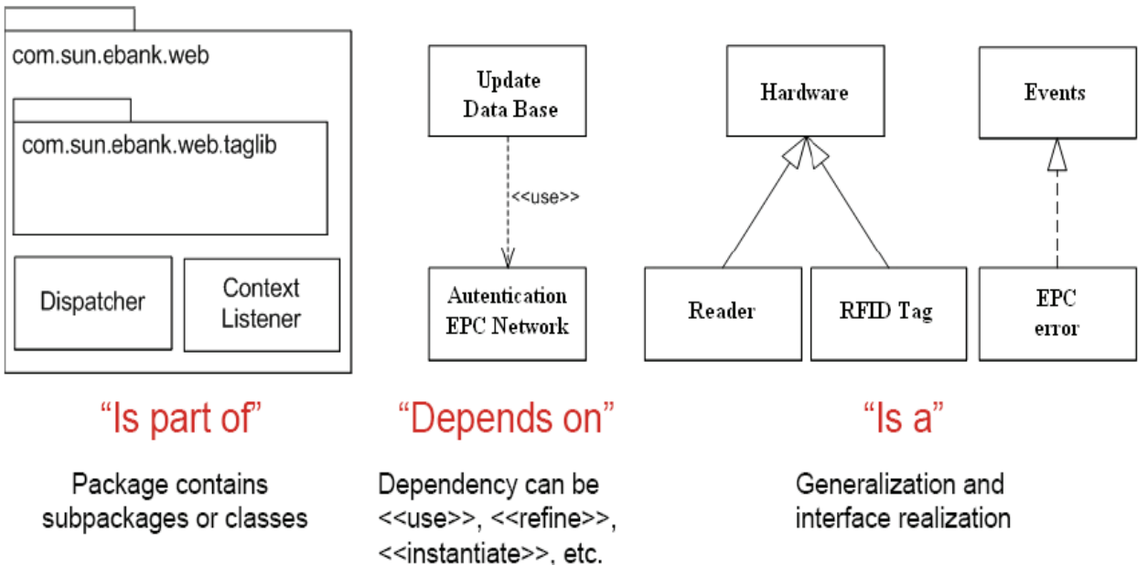

Fig. 6. UML Relations between modules

In deployment view our goal is to describe the hardware structure which the systems are running. Thus, is possible to verify the interconnection between EPC Information Services,

to analyse the performance of the EPCglobal Network, security, and access control to data bases. The UML 2.0 define the deployment diagram with goal of shows the physical deployment of the system, such as the computers, and devices (nodes) and how connect to each other.

Finally, the data view can be used to describe the data bases modelling and their relationship. The goal this view is to improve performance and adaptability of the systems, and to avoid redundancy and enforce consistency. In RFID-based supply chains context the data view is stronger used to represent the data bases that store information about each RFUD tag. The UML diagram that better show the data view representation is the class diagram. However, this view can also be represented entity-relationship diagram.

6.4 Architecture documentation

After defining the view, the domain designer will make the architecture documentation, especially, information that will be applied to more than a vision. In this sense, we define a template with goal of to assist architecture documentation.

Architecture Documentation
1. Guidelines
Describe the way that the architecture documentation is organized, including the use this document in Supply Chain.
2. Design Information
Show design information as, for example, EPC version, previous and auxiliary documents, design members, and goals in general lines.
3. Domain Information
Describe the domain that will project their quality attributes, functional and non-functional requirements with major relevance for supply chain designers.
4. Views Documentation
4.1 Name
4.2 Graphic Representation
4.3 Elements Description
4.4 Relationship of views
4.5 Others information
5. Relation between Analysis and Design
Show which requirements described in analysis phase are in architecture
6. Glossary
Glossary of the system and acronyms

7. The domain implementation step

The last phase of the domain engineering process for RFID-based systems development in supply chain is the Domain Implementation. In concordance with (Pohl et al., 2005), the goal of this step is to provide the implementation and documentation of the reusable assets described in previous step. The activities defined in this chapter for domain implementation step are in concordance with component-based development methods and software reuse processes, among this process are UML Components (Cheesman & Daniels, 2000) and Catalysis (D'Souza & Wills, 1998). The following sections show activities of the domain implementation.

7.1 Component implementation

In this activity, the software engineer, based on requirements, implements the software components through a set of well defined sub-activities. The approach is intended to be used in the scope of domain engineering, and therefore it depends on assets developed in domain analysis (feature model, requirements, domain use case model) and domain design (domain-specific software architecture, component specifications).

This activity is divided into two sets of sub-activities, each one with a different purpose. Sub-activities 1 to 4 deal with the *provided* services, i.e. when the software engineer wants to implement a component to be reused. Sub-activities 5 to 7 deal with *required* services, i.e. when the software engineer wants to reuse services from existent components. The first sub-activity is to describe the component, providing general-purpose information, such as the component vendor, version, package, among others. This information may be used to identify a component, an important issue when components are stored in a repository, for example.

In this second sub-activity, the software engineer should *specify the interfaces*. However, as mentioned before, the domain implementation method depends on artefacts developed in domain analysis and design, such as the domain-specific software architecture and component specifications. These artefacts already contain the interface specification, and so the software engineer only needs to review and refine them, if necessary. In the third sub-activity, the goal is to *implement the services* defined in the previous sub-activity, using any implementation technology, as well as the code to *register these services to be used by other components*, if a dynamic execution environment is used. In fourth sub-activity, which concludes the provided side of the component, the goal is to *build and install the component*. According to the implementation technology used, this involves compiling and packaging the component in a form that is suitable to be deployed in the production environment.

Sub-activities 1 to 4 deal with the *provided* side of a component. In order to implement the *required* side, three sub-activities should be performed: First, the software engineer needs to describe the component that will reuse other services. This is similar to first sub-activity, but with the focus on the services that are required. In this sub-activity, the code that accesses the required services is implemented. Here, different techniques can be employed, such as the use of adapters, wrappers, or other ways to implement this access. The main goal of this sub-activity is to provide low coupling between the required service and the rest of the code, so that it can be more easily modified or replaced. The last sub-activity corresponds to *building and installing the component* that reuses the services, which is similar to fourth sub-activity. Although these two sets of sub-activities (1-4 and 5-7) are focused on different

aspects, in practice they will be present in most components, since normally each component has *both* provided and required interfaces.

7.2 Component documentation

When a component is designed and implemented, the developer has clearly in mind some test scenarios and specifics set of the use cases. Thus, case the client does not encompass the component goals, it will be used incorrect way. In this sense, the component documentation is presented by Sametinger (1997) as "*a direct path for information from author to customer, transfers knowledge efficiently. It is one of the most important ways to improve program comprehension [and reduce] software costs*"(Sametinger 1997). This way, Kotula (Kotula 1998) presents thirty nine interrelated patterns as solution for documentation of quality. It is grouped in six categories: (i) *Generative Patterns*: which describe high-level, pattern-creating pattern, (ii) *Content Pattern*: which describe the material that must be included in the documentation, (iii) *Structure Patterns*: which describe how the documentation must be organized, (iv) *Search Patterns*: which discuss the facilities needed to find specific information, (v) *Presentation Patterns*: describing how the documentation should be presented graphically; and (vi) *Embedding Patterns*: which provide guidelines for how to embed documentation content within source code.
Other the hand, (Taulavuori et al., 2004) says that "*definition of the documentation pattern is not sufficient for the adoption of a new documentation practice. An environment that supports the development of documentation is also required*". This way, Taulavuori et al., (2004) provide guidelines concerning how to document a software component. After to analyse patterns defined by Kotula (1998) and component documentation in the context of software product lines, described in (Taulavuori et al., 2004), this chapter defines the following template for component documentation.

Component Documentation
1.General Information
1.1 Name
Should be well defined and describe the component
1.2 Type
Expresses the way the component is intended to be used
1.3 Goal
Describe the relation with the RFID technology present in supply chain
1.4 History
Describes the life cycle of the component.
2. Interfaces
2.1 Required Interfaces
The interface information is here defined as including the interface name, type, description, behaviour and interface functions.

2.2 Provided Interfaces
The same that required interfaces
3. Standards
3.1 Protocol
Describe the interaction between two components needed to achieve a specific objective
3.2 Standards
What EPCglobal Network standards the component is using? What standards are necessaries? Standards can restrict the compatibility, structure and functionality of components.
4. Technical Details
The Technical Details describes details the of design and implementation component.
4.1 Development Environment
Defines the environment in which the component has been development.
4.2 Interdependencies
Describes component's dependency on the components
4.3 Prerequisites
Defines all the other requirements that component may have to operate.
4.4 Implementation
Implementation includes composition, context, configuration, and interface implementation. Composition information describes the internal structure of the component, which can be derived from the component's class diagram. The component's class diagram must be included, if possible, as well as the classes, operations and attributes.
4.5 Restrictions
Should describe all the items that will limit the provider's options for designing or implementing the components.
5. Information Non-functional
5.1 Modifiability
Define as the component can be adapted in new supply chains.
5.3 Expandability
Describes how new is often qualified only for OCM components.
5.3 Performance
Performance is a quality attribute that measures the component.. The measurements are

size of the component, prioritisations of events, and utilization in RFID systems.
5.4 Security
Define strategies to combat hackers. Including cryptography in RFID tags level.
6. General Information
6.1 Installation guide
Defines the operations that must be performed before the component can be used.
6.2 Support
Customer support includes the name of the contact person and an address or a phone number where the customer can get help.

8. Conclusion

As widely discussed in this chapter, the reuse process present gaps and lack of details in key activities such as, for example, domain engineering. In this sense, we believe that our approach can be useful to reduce the gaps and lack of details among the steps, and presenting a domain engineering process for RFID-based systems development presents in supply chain domain. This work can be seen as initial climbing towards the full vision for an efficient domain engineering process and interesting directions remain to improve what started and to explore new routes. Thus, the following issues should be investigated as future work: (i) Other RFID Standards, (ii) Other Directions in Software Architecture, for instance, Service-oriented or Model-Driven and (iii) Architecture Documentation.

9. References

Alexander, C. (1977). A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977, pp. 1216

Almeida, E. S. et al., (2005). A Survey on Software Reuse Processes, *IEEE International Conference on Information Reuse and Integration,* pp. 66-71, Nevada, USA, August, 2005, Las Vegas

Almeida, E. S. et al., (2006). The Domain Analysis Concept Revisited: A Practical Approach, *9th International Conference on Software Reuse,* Lecture Notes in Computer Science, Torino, Italy, June, 2006, pp. 43-57

Almeida, E. S. (2007). RiDE – The RiSE Process for Domain Engineering, Ph. D. Thesis, Federal University of Pernambuco, Recife, Brazil, 2007

Arango, G. (1989). Domain Analysis: from art form to Engineering Discipline, *5th International Workshop on Software specification and design,* Pennsylvania, USA, May, 1989, Pittsburgh

Armenio, F et al., (2007) *The EPCglobal Architecture Framework,* EPCglobal Final Version, pp. 7, September 2007

Atkinson, C.; Bayer, J. & Muthig, D. (2000). Component-Based Product Line Development: The KobrA Approach, *First Software Product Line Conference,* Kluwer International Series in Software Engineering and Computer Science, pp. 19, Colocado, USA, August, 2000, Denver

Bauer, D. (1993). A Reusable Parts Center, *IBM Systems Journal,* Vol. 32, No. 04, pp. 620-624, September, 1993

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement, *IEEE Computer,* Vol. 21, No. 05, pp. 61-72, May, 1988

Bosch, J. (2000). Design and Use of Software Architecture, Addison-Wesley, 2000, pp. 354

Campos, L. B. & Zorzo, S. D. (2007). A Domain Analysis Approach for Engineering RFID Systems in Supply Chain Management, *IEEE International Conference on System of Systems Engineering,* Texas, USA, pp. 165-171, April, 2007, San Antonio

Cheesman, J. & Daniels, J. (2000). UML Component A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2000, pp. 208

Clements, P & Northrop, L. (2001). Software Product Lines: Practices and Patterns, Addison-Wesley, 2001, pp. 608

Clements, P. et al., (2004). Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2004, pp. 512

Czarnecki, K & Eisenecker, U. W. (2000). Generative Programming: Methods, Tools, and Applications, pp. 832, Addison-Wesley, 2000

Dikel, D. (1997). Applying Software Product-Line Architecture, *IEEE Computer,* Vol. 30, No. 08, pp. 49-55, August, 1997

D'Souza, D. F. & Wills, A. C. (1998). Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998, pp. 816

Endres, A. (1993). Lessons Learned in an Industrial Software Lab, *IEEE Software,* Vol. 10, No. 05, pp. 58-61, September, 1993

Engels, D (2003). The use of the Electronic Product Code, *White Paper of the Massachusetts Institute of Technology* (MIT), pp. 03, February 2003.

EPCglobal (2005). Object Naming Service (ONS) Version 1.0, EPCglobal Ratified Specifications, pp. 09, October, 2005

Ezran, M.; Morisio, M & Tully, C. (2002). Practical Software Reuse, pp. 374, Springer

Frakes, W. B. & Isoda, S. (1994). Success Factors of Systematic Software Reuse, *IEEE Software,* Vol. 12, No. 01, pp. 15-19, September, 1994

Frakes, W. B. & Kang, K. C. (2005). Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering,* Vol. 31, No. 07, pp. 529-536, July, 2005

Gamma, E. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, pp. 395

Gomaa, H. (2005). Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley, 2005, pp. 701

Griss, M. L. (1994). Software Reuse Experience at Hewlett-Packard, *16th IEEE International Conference on Software Engineering,* pp. 270, Italy, May, 1994, Sorrento

Griss, M. L. (1995). Making Software Reuse Work at Hewlett-Packard, *IEEE Software,* Vol. 12, No. 01, pp. 105-107, January, 1995

Harrison, M. (2003). EPC Information Service – Data Model and Queries, White paper of the Massachusetts Institute of Technology (MIT), pp. 03, October, 2003

Jacobson, I.; Booch, G. & Rumbaugh, J. (1999). The Unified Software Development Process, Addison-Wesley, 1999, pp. 463

Joos, R. (1994). Software Reuse at Motorola, *IEEE Software,* Vol. 11, No. 05, pp. 42-47, September, 1994

Kang. K. C. et al., (1998). FORM: A Feature-Oriented Reuse Method with domain-specific reference architectures, *Annals of Software Engineering Notes,* Vol. 05, No. 00, January, 1998, pp; 143-168

Kang, K. C.; Lee, J. & Donohoe, P. (2002). Feature-Oriented Product Line Engineering, *IEEE Software,* Vol. 19, No. 04, pp; 58-65, July/August, 2002

Kazman et al., (2005). A Basis for Analyzing Software Architecture Analysis Methods, *Software Quality Journal,* Vol. 13, No. 04, pp. 329-355, December, 2005

Kotula, J. (1998). Using Patterns To Create Component Documentation. *IEEE Software,* Vol. 15, No. 02, pp. 84-92, March/April, 1998

Kruchten, P.; Obbink, H. & Stafford, J. (2006). The Past, Present, and Future of Software Architecture, *IEEE Software*, Vol. 23, No. 02, pp. 22-30, March/April, 2006

Lee, K. & Kang, K. C. (2004). Feature Dependency Analysis for Product Line Component Design, *8th International Conference on Software Reuse,* Madri, Spain, July, 2004, pp. 69-85

Morisio, M; Ezran, M & Tully, C. (2002). Success and Failure Factors in Software Reuse, *IEEE Transactions on Software Engineering*, Vol. 28, No. 04, pp. 340-357, April, 2002

Neighbors, J. M. (1989). Draco: A Method for Engineering Reusable Software Systems, in Software Reusability Volume I: Concepts and Models, T. Biggerstaff, A. Perlis, 1989, pp. 425

Osterweil, L. (1987). Software Process are Software too, *9th International Conference on Software Engineering*, pp. 02-13, California, USA, March/April, 1987, Monterey

Papazoglou, M. P. & Georgakopoulos, D. (2003). Service-Oriented Computing, *Communications of the ACM,*Vol. 46, No. 10, pp. 25-28, October, 2003

Pohl, K.; Bockle, G. & van der Linden, F. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, 2005, pp. 467

Pressman, R. S. (2005). Software Engineering: A Practitioner's Approach , pp. 880, McGraw-Hill, 2005

Prieto-Diaz, R. (1990). Domain Analysis: An Introduction, *ACM SIGSOFT Software Engineering Notes,* Vol. 15, No. 02, pp. 47-54, April, 1990

Rine. D. C. (1997). Success Factors for Software Reuse that are Applicable Across Domains and Businesses, *ACM Symposium on Applied Computing*, pp. 182-186, California, USA, March, 1997, San Jose

Rothenberger, M. A.; Dooley, K. J.; Kulkarni, U. R. & Nada, N. (2003). Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices, *IEEE Transactions on Software Engineering,* Vol. 29, No. 09, pp. 825-837, September, 2003

Sabbaghi, A. & Valdyanathan, G. (2008). Effectiveness and Efficiency of RFID Technology in Supply Chain Management: Strategic values and Challenges. *Journal of Theoretical and Applied Electronic Commerce Research,* Vol. 03, No. 02, August 2008, pp. 71-81, ISSN 0718-1876 Electronic Version

Sametinger, J. (1997). Software Engineering with Reusable Components, Springer-Verlag, 1997, pp. 275

Schmidt, D. C. (2006). Model-Driven Engineering, *IEEE Computer,* Vol. 39, No. 02, pp. 25-31, February, 2006

Shaw, M & Garlan, D. (1996). Software Architecture: Perspective on an Emerging Discipline, Prentice Hall, pp. 242

Simos, M. et al. (1996). Organization Domain Modeling (ODM) Guidebook, Version 2.0, *Technical Report,* June, 1996, pp. 509

Sommerville, I. (2006). Software Engineering, pp. 840, Addison Wesley

Supply-Chain Council (1997). New group aims to improve supply chain integration, Purchasing, pp. 8-32, Vol. 123, No. 6

Svahnberg, M; van Gurp, J & Bosch, J. (2001). On the Notion of Variabilities in Software Product Lines, *Working IEEE/IFIP Conference on Software Architecture,* Amsterdam, Netherlands, August, 2001, pp. 45-54

Taulavuori, A.; Niemela, E. & Kallio, P; (2004). Component Documentation – a key issue in software product lines, *Journal Information and Software Technology,* Vol. 46, No. 08, pp. 535-546, June, 2004

Tracz, W. (1995). DSSA (Domain-Specific Software Architecture) Pedagogical Example, *ACM SIGSOFT Software Engineering Notes,* Vol. 20, No. 03, pp. 49-62, July, 1995

Weiss, D. M. & Lai, C. T. R. (1999). Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999, pp. 426

Winter, M.; Zeidler, C. & Stich, C. (2002). The PECOS Software Process, *Workshop on Component-based Software Development, 7th International Conference on Software Reuse*, pp. 07, Texas, USA, April, 2002, Austin

Supply Chain the Way to Flat Organisation

Edited by Julio Ponce and Adem Karahoca

ISBN 978-953-7619-35-0
Hard cover, 436 pages
Publisher InTech
Published online 01, January, 2009
Published in print edition January, 2009

With the ever-increasing levels of volatility in demand and more and more turbulent market conditions, there is a growing acceptance that individual businesses can no longer compete as stand-alone entities but rather as supply chains. Supply chain management (SCM) has been both an emergent field of practice and an academic domain to help firms satisfy customer needs more responsively with improved quality, reduction cost and higher flexibility. This book discusses some of the latest development and findings addressing a number of key areas of aspect of supply chain management, including the application and development ICT and the RFID technique in SCM, SCM modeling and control, and number of emerging trends and issues.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen