Domain Analysis and Framework-based Software Development

Andrea Valério\textsuperscript{1}, Giancarlo Succi\textsuperscript{2}, Massimo Fenaroli\textsuperscript{3}

\textsuperscript{1} Software Production Engineering Lab (LIPS), Department of Communication, Computer and System Sciences, University of Genova, Genova (GE), Italy
e-mail: Andrea.Valerio@dist.unige.it

\textsuperscript{2} Department of Electrical and Computer Engineering The University of Calgary, Calgary, Alberta, Canada
E-mail: Giancarlo.Succi@enel.ucalgary.ca

\textsuperscript{3} Thera S.p.A., Brescia (BS), Italy
E-mail: theralab@ns.numerica.it

Abstract

Domain Analysis is the process that identifies the relevant objects of an application domain. The goal of Domain Analysis is Software Reuse. The higher is the level of the life-cycle object to reuse, the larger are the benefits coming from its reuse, the harder is the definition of a workable process. Frameworks are excellent candidates for Domain Analysis: they are at a higher level than code but average programmers can understand them. This paper presents the main features of Sherlock, a domain analysis process for the extraction of reusable frameworks, and discusses the impact that Sherlock has on the software process, in particular concerning reuse practices. Sherlock is based on both FODA and Proteus domain analysis techniques. The input of Sherlock is an informal description of the domain based on other domain applications, literature, user requirements, and interviews with domain experts. The output of Sherlock is a set of architectural models (the domain frameworks and patterns), taxonomies of permanencies and variabilities, objects models focusing on the common domain objects. The introduction of Sherlock in the software process requires the definition of new roles and has a valuable impact on reuse practices. We present in this paper a case study: the application of Sherlock at Thera S.p.A., a software company located in Brescia, Italy.

1. Introduction

Nowadays, software reuse is considered as one of the most promising approach in software engineering. Software reuse provides the foundation for improving the way software is developed and supported over its life-cycle. Its potential benefits range from decreased development time and increased product quality to improved reliability and decreased maintenance costs. However, these benefits are fully exploitable only if reuse practices are integrated and supported in the software development process.

Reuse fosters modularity and interoperability of applications, even more if integrated with object-oriented paradigms: both reuse and object-oriented paradigms are centered on the concept of asset, i. e. an entity that can be completely and independently modeled and formalized in a given context for reuse purposes.

Frameworks are an interesting technology based on object-oriented methodology and fostering reuse practices. A framework can be defined as "a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact", or, with another definition, "a skeleton of an application that can be customized by an application developer" [Johnson97]. As pointed out By Johnson, there is an implicit difficulty in defining what is a framework. Sometimes, speaking about software reuse, frameworks are compared to generic components with a high degree of abstraction, but frameworks can probably be best described as the context that defines specific 'pattern' of communication and cooperation among software components. Frameworks are tightly coupled to patterns in the way that they could represent the instantiation of a solution to a problem in a defined context. In particular, instantiation means that frameworks could be the connecting ring
between design and code. Frameworks, like software components and other assets, can be packed and made available for reuse inside the software life-cycle.

Domain analysis processes existing software application in order to extract and packet reusable assets. Systematic reuse requires an understanding of previous works, in particular a major problem concerns the creation of assets that can be reused in a context different from that where they have been. In this view, domain analysis is a fundamental activity integrated in a software process based on reuse.

Several approaches to domain analysis exist, but in general they are not widespread. One reason could be that sometimes they are too difficult and rigid; another reason could be that they target assets that do not have a high reuse potential. Almost none of them is specifically target to design frameworks.

In this work it is presented a new approach to domain analysis called Sherlock, based on PROTEUS [PROTEUS94] and on FODA [FODA90]. Sherlock prescribes a three steps process: domain characterization, data analysis and modeling, and definition of a reusable architecture; we outline Sherlock and its main features in Section 3. Sherlock is specifically targeted to framework extraction and reuse: it delivers reusable frameworks extracted from the domain, modeling commonalities of the domain and encapsulating variation possibilities, which are logically correlated in a general domain architecture; we describe these aspects in Section 4. Sherlock, when introduced and used inside the software process, play a fundamental role in the analysis and design phases (aimed to the production of reusable framework and assets), and fostering reuse practices. We think that one of the strongest point of Sherlock is that it has proven to work; we describe this success story in Section 5; it concerns the production of a graphic interface modules. We end the paper with some consideration on the effects of Sherlock on the software process and presenting the future works we plan to carry out.

2. Domain Analysis - The State of the Art

Domain analysis can be defined as: "a process by which information used in developing software systems is identified, captured and organized with the purpose of making it reusable when creating new systems" [Prieto-Diaz90]. During software development, different information is produced, and the software product delivered is only a part of this heap of data. One of the main goals of domain analysis is to analyze all these information aiming to exploit and reuse most of them in future software development projects. Following this view, domain analysis fosters software reuse in the sense that it supports the identification and definition of information and components that can be reused in applications and in contexts different from the ones for which they were originally conceived.

Domain analysis is an activity occurring prior to system analysis. It aims to identify features common to a domain of applications, selecting and abstracting the objects and operations that characterize those features. The first definition of domain analysis has been introduced by Neighbors as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain" [Neighbors81]. In his work, Neighbors draws an analogy between domain analysis and system analysis: system analysis is concerned with the specific aspects in a particular system, while domain analysis is concerned with objects and operations in a set of systems in a defined application area. The generalization of the systems in an application domain aims to define domain models that transcend specific applications. Generally speaking, domain analysis should support extraction, organization, analysis and abstraction, understanding, representation and modeling of reusable information and assets from the software process [Prieto-Diaz90]. In these last years these concepts have been exploited and different methodologies for domain analysis have been proposed. Readers interested in a more comprehensive survey on Domain Analysis should consult [Arango 93].

Considering the activities which are shared by the different Domain Analysis methods, it is possible to define a general model for the domain analysis process which can help in comparing the different methodologies (in [Arango93] it is presented an extended discussion on a Common Domain Analysis Process). Along the lines of
[Arango 93], the general model can be structured into four main phases, each phase constituted by different activities:

1) **Domain Characterization and project planning**: the first step of every domain analysis method is a preparation activity, which aims to collect the minimum information concerning the problem that allows to decide if it is worth to deal with it and try to solve it, or if it is not feasible to go on with the process. The main activities that have to be carried out during this phase are:
   a) Business Analysis;
   b) Feasibility Analysis;
   c) Domain Description;
   d) Project planning and resource allocation.

2) **Data Analysis**: the information necessary to the analysis are collected and organized, then the analysis exploits domain commonalities and variations. The main activities prescribed are:
   a) Data Organization
   b) Data Exploitation

3) **Domain Modeling**: the purpose of the modeling phase is to complete the previous analysis step building suitable domain models of the domain, specifically:
   a) Modeling commonalities aspects in the domain;
   b) Refining domain models encapsulating variations possibilities;
   c) Defining frameworks and general architecture for the domain;
   d) Describing the rationale beneath domain models and tracing technical issue and relative decisions made in the analysis and modeling process.

This phase can be considered the core activity aiming to produce reusable assets, such as components, framework and architectures. The difference between domain modeling and system modeling lies in the target chosen: for system modeling it is the specific software system that has to be built, while for domain analysis it is a class of similar systems in a specific application domain.

4) **Evaluation**: its purpose is to verify the results of each step of the domain analysis process, identifying possible errors done in building the model, and to validate the results against requirements and user expectations.

The general model just outlined represents a possible common process for domain analysis methodologies, summarizing the main activities shared by the different methods. It is noticeable that each methods has its own peculiarities due to the specific problem that has to be solved and to the approach, such as problem driven or application driven, or else reuse-oriented or knowledge-representation oriented. Wartik and Prieto-Diaz have proposed an interesting analysis based on set of criteria suitable for comparing reuse-oriented domain analysis approaches: in their work it is highlighted that “comparing different domain analysis approaches requires considering the context in which an approach will be used” (see [WP92]). In particular, they propose five context factors: software process needs, existing software base, business objectives, state of domain knowledge and intended use of information repositories.

Some general comments can be done considering the different Domain Analysis approaches proposed in literature:

- No methodology addresses the problem of extracting reusable frameworks;
- In the preparation phase preceding data analysis, the domain scoping activity is weak: business analysis, feasibility study, project planning, and resource allocation are often performed just using personnel capability and experience. It lacks a clear description of the tasks that have to be undertaken and suitable guidelines that support personnel in carrying on these activities.
- The approaches proposed are static, in the sense that their process is mostly linear and driving to a ‘one-shot’ result. Evolutionary behavior is a fact of real-life domains, and domain analysis should be able to identify and describe possible evolution of a domain, guiding the user through an interactive and cyclic process aiming to upgrade and to improve domain understanding and representation at each successive step, maintaining domain models ‘alive’ and following the dynamic changing boundaries of the domain.
Customization is becoming one of the most important characteristics for a successful software product: while software reuse and domain analysis connected to it foster standardization and mass production, a software firm can be successful only if it is able to satisfy specific user needs, exploiting the added value of product variation and customization. In this view, domain analysis should focus on the identification of variants and possibilities for later customization and specialization, encapsulating these aspects inside domain models (such as frameworks and architecture) which can be composed into customized software products that meet user needs.

Domain analysis should be part of the software process, using the same techniques and similar practices when possible, sharing human resources. Domain analysis has to support software development without overloading it with further activities, practices and deliverables. Products delivered by domain analysis should be homogeneous with assets usually circulating in the software process.

Last but not least, as indicated by Arango in [Arango93], a useful method should be reliable, giving confidence to its user that applying the method the foreseen results will be achieved, and repeatable, assuring that different people applying the same method in similar contexts will yield similar results. Moreover, it should provide verification procedures and validation criteria in order to evaluate if a user is doing the right things towards its goals and, once the results have been achieved, to decide whether or not it has accomplished its objectives. Considering these issues for domain analysis methods previously presented, it results that in general they are weak and inadequate as regarding reliability and replicability, and only a few explicitly prescribe some verification and validation activities.

Walking side by side to software reuse, the emphasis in domain analysis has moved from the analysis of code to the analysis of every kind of information produced in the software process, with the goal to identify and define high level reusable artifacts, such as frameworks and architectures. In this view, the domain analysis process became a fundamental part of a global software engineering process whose purpose is to produce new application reusing past components, frameworks and information and aggregating them following the model proposed by a general domain architecture [BGKLRZ97].

3. Sherlock: a Domain Analysis approach for component-based software development

As a result of the analysis of different domain analysis approaches, we have identified a set of desirable features that a robust domain analysis methodology should have:

- an effective domain analysis methodology has to be adaptable to the software process of the organization where it is being introduced; its use inside the usual process activities should be immediate and intuitive;
- the methodology should be repeatable in different projects in the organization with different resources, in particular people;
- the methodology has to foster framework (and component) development and reuse, since frameworks seem the best candidate for software reuse [FS97];
- during the application of the methodology, it should be possible to monitor and to control the activities and the results, giving confidence on its reliability;
- models and deliverables produced during the analysis should be verified and validated with adequate procedures; for example, domain experts verify the consistency of the domain models; software engineers or the analysts validate the domain models applying them to new applications or problems that have not already been considered in the domain; well known techniques, as formal reviews, are used to verify if products quality is improved.

These requirements have pushed the development of Sherlock, a domain analysis approach in the context of an incremental software process based on object oriented techniques.

In particular, the following characteristics describe the reference development environment we have considered for the definition of Sherlock:

- An iterative or incremental development process (being the waterfall model a special case of the former)
- Object Oriented Analysis and Design: Sherlock uses the UML notation, but it can be easily ‘ported’ to Booch or OMT notations
• Object Oriented Programming, either using an Object Oriented Language, such as C++ or Java, or using an imperative language, such as C, Cobol, or Pascal, with an Object Oriented approach

3.2 The domain analysis methodologies taken as baseline for Sherlock

PROTEUS and FODA are the starting references of Sherlock: they are general approaches with a rather detailed and accessible documentation. Other methodologies proposed refer to particular aspects that do not match our requirements, even though they could be more powerful if applied in the specific context for which they have been devised.

The domain analysis methodology proposed by PROTEUS is the best match for our context: an incremental software process based on object oriented techniques. The PROTEUS methodology is divided into three iterative phases: domain description and qualification, domain requirements and architectural modeling, model validation. PROTEUS applies, for the analysis and modeling activities, the Object Modeling Technique (OMT) written by J. Rumbaugh and others, representing commonalities and variants with the concepts (common to most of the object oriented techniques) of: constraints and associations, generalization and specialization, abstraction and aggregation, multiplicity and metadata/metamodel. The principal deliverable produced are: the Feasibility Report and the Qualified Domain Description (end of first phase), and the Domain Requirements Model and the Domain Architectural Model (derived from the OMT Object, Dynamic and Functional Model during the second phase).

We refer also to FODA, mainly because it is interesting its approach that focus on user-visible aspects of software systems and some useful concepts and deliverables. FODA defines three main phases: context analysis, domain modeling and architecture modeling. The primary focus of the method is the identification of distinctive features of applications in a domain (user-visible aspects of a system in an application domain), and its approach to the analysis and characterization of systems in a domain can be considered functional. FODA is based on well established software engineering techniques, such as entity-relationship-attribute models, data-flow diagrams and finite-state machine models, but these techniques are scarcely compatible with object oriented methods. FODA description does not cover non-technical issues related to domain analysis and the description of the third phase concerning the identification of domain architecture is not fully exploited.

Sherlock takes from PROTEUS the general process based on object oriented techniques, and from FODA the domain characterization (specifically, the gathering of domain requirements and user expectations) and the documentation strategy. Both PROTEUS and FODA aims to produce a domain architecture: Sherlock expands this activity introducing the related concepts of frameworks and general domain architecture, delivered as reusable assets in the domain.

3.3 The main features of Sherlock

The main features of Sherlock are the following:

• the main products of the analysis are reusable frameworks and a general domain architecture which define the typical structure of the applications domain and synthesize the knowledge and the experience gathered in the domain
• domain scoping is conducted using structured documents that describe the functionalities which have been formalized capturing applications' requirements and users' needs, in a way similar to the FODA feature, considering the user point of view
• Sherlock follows the main concepts proposed by the PROTEUS domain analysis methodology, substituting OMT with UML [UML97] for developing domain models and for identifying the commonalities in the domain;
• the extraction of the variable aspects is parallel to the definition of domain models and is based on a custom variability classification schema [Schmid97];

Sherlock introduces into the domain characterization phase a preliminary activity of feasibility study, cost-benefit and risk analysis. They play the role of preface tasks whose positive outcome is a necessary condition for
subsequent development activities, in particular for starting the domain analysis process and for allocating adequate resources to it. During the analysis, a regular monitoring activity supervises the evolution of the process and identifies the required corrective actions. In the current version of the methodology, these preliminary activities have not yet been specified fully: they are performed following the traditional techniques; however, Sherlock introduces a verification and validation activity that is spread throughout the whole process and controls of the state of the work, also from the point of view of marketing and management department.

Sherlock is divided into three main phases:

- **Domain characterization**: during this phase all the information available concerning the target domain are collected and organized; a domain terminology vocabulary is written containing the definition and the meaning of each term found that is related to the domain. These information are used to characterize the domain, identifying its boundaries (and implicitly what is in and what is out). The context of the analysis process is defined by structured descriptions of the functionalities that formalize requirements and needs in the users’ point of view.

- **Data analysis and modeling**: the domain is analyzed to extract commonalties and variants. Sherlock uses an UML like structure for the design and the construction of domain models. Variable aspects are collected and classified using forms specifically provided. Deliverable produced will be supported by an issue/decision description document (derived from the homonymous document prescribed by FODA), that records and explains problems encountered and solutions adopted, tracing the motivation for the technical decisions taken.

- **Definition of a reusable architecture**: in this phase structured knowledge and information acquired from previous activities are employed to refine domain object oriented models, in particular domain frameworks, and to develop a general domain architecture (pinpointing the components structure and relations) that can be reused for the production of new software systems in the domain analyzed. These phases are integrated with a continuous activity of verification and validation of the current state of the process and of the products delivered. This introduce an interactive character in the process, allowing to detect and correct possible errors made in the analysis and to evolve the domain description following the dynamics and the evolution of the applications domain in the software process.

### 3.4 Roles in Sherlock

An important task performed during the first step of the Sherlock process regards the allocation of personnel to the project, identifying roles and responsibilities for each activities that has to be carried out. The domain analysis activities are performed by a domain analysis staff composed by people playing different roles:

- The **domain analyst** supervises the analysis and modeling activities, organizing the work-team and managing each single phase of the process. He is in charge of gathering all the information needed for the analysis and, in collaboration with domain experts and software engineers, he analyses the information and to define required models. The presence of the domain analysts is continuous along the whole process and guarantees continuity to the activities.

- The **domain expert** is one of the main sources of knowledge and expertise concerning the domain considered and helps to frame applications inside the boundaries of the domain. Besides, he identifies customer requirements and expectations, and supports the domain analyst in gathering and analyzing information on the domain. He participates to the domain characterization activity, giving its support for identifying models and defining the general architecture, in particular regarding the extraction of commonalties and variants.

- The **software engineer** is the expert for the abstraction and modeling techniques applying UML. He supports the analysis of applications inside the domain, their modeling and is mainly responsible for the construction of object oriented models and of the general domain architecture.

### 4. The extraction of Reusable Frameworks in Sherlock

One of the core activities in Sherlock is the identification and the production of reusable frameworks and components. The high level information concerning the organization of components and relationships among the framework structures is encapsulated in a general architecture of the domain. Reusable artifacts extracted from
the analysis process include also knowledge and experience derived from the information collected in the domain.

The input resources for this phase are primarily the object-oriented models and the document that classifies the variants. Furthermore, the issue/decision document is helpful in considering past binding technical decision and in tracing important issues encountered in the analysis and modeling process, solutions proposed, and decisions taken. As a result, this phase produces different architectural models, i.e. reusable frameworks and a general architecture of the domain.

The term ‘architecture’ and ‘framework’ can have multiple meanings; in this context architecture means a representation of the categories and frameworks that constitute a software system and of the relationships inside it; a architecture encapsulates the structure and the organization of common applications in the domain. A category is a set of classes and objects that are strongly cohesive internally but loosely coupled with external classes. A framework is a set of components (classes) with specific interactions among them, and encapsulating an abstract and generalized solution to a family of related problems in an application domain; there are three kinds of frameworks:

- white-box frameworks: they are characterized by the presence of classes that have to be specialized through inheritance, redefining abstract or not significative methods;
- black-box frameworks: they are characterized by the presence of ready-to-use components that have to be assembled composing communicating objects;
- gray-box frameworks: they are in between the previous kinds of frameworks.

The extraction of reusable frameworks is the abstraction of object oriented domain models, to obtain high level architectural components (as compared to classes) and to represent domain characteristic features as interactions and relationships among modular reusable components. These artifacts embody a general solution to a specific domain issue, as well as control mechanism for dealing with variability and flexibility in different reuse contexts. The activities that are performed during the framework extraction are:

1) **Identification**: inside the object oriented model of the domain, we identify sets of classes that are strongly coupled among themselves but loosely coupled to other classes in the model: these sets of classes, grouped and identified as self-standing entities, are the candidates for the identification of frameworks. To group the classes into categories we reorganize the object-oriented models of the applications inside the domain structuring them in more levels of the representation.

2) **Refinement**: the groups of classes identified in the previous step are refined considering possible variants extracted during the analysis of the application domain. For each candidate framework, all the possible variants are considered; this activity is performed following the indication given in the variability classification document. The structure of the candidate framework is modified with the introduction of technical solutions that allows the requested kind of variability of the structure (modification and specialization). During this activity, it is possible that the object oriented models undergo structural modifications, primarily concerned with the internal and external organization and the relationships among classes: most of the times, when it is feasible, these changes are undertaken following the prescriptions contained into selected design patterns. This modification and refinement process driven by appropriate design patterns, seen as knowledge and expertise formalized model for dealing with specific problems, aims to reuse previous successful experiences in solving a similar problem. Design patterns, extracted by literature or devised from domain applications and past projects, can be seen as the rationale behind frameworks, which in turn assume the role of concrete reusable modules.

3) **Representation** of the domain as a collection of interconnected entities: the frameworks identified in the previous steps are organized into a high level view, i.e. a general domain architecture. In this phase the focus shifts from single frameworks extraction to the description of the meta-information embodied in the cooperation and relationships among frameworks aiming to act as a whole application. Frameworks are now the bricks used to build a system that satisfies an underlying structure. In the same way as we have done with frameworks, one or more diagrams are produced, possibly implementing the prescriptions of an architectural design pattern, representing the relationships and the cooperation among frameworks.
5. A case study: the application of Sherlock in Thera

The case study presented regards the development of a graphic interface module for information systems of business and management units. It is a project conducted during the first 5 months of 1997 inside Thera S.p.A., one of the major Italian firms in information system production for private and public bodies. The resources consumed have been 20 person months. The graphic interface module delivered at the end of the project is currently used inside different software systems.

5.1 Definition of the context

Considering graphic interfaces, the work has been focused on the creation of a generalized architecture that allows to automatically sense data acquisitions and checks values introduced as defined in the classes of the Business Model, but that to the same time guarantees uncoupling between the User Interface and the Business Model (in other words, the User Interface 'knows' the classes of the Business Model, but the opposite is not true).

A Business Model is an object-oriented model of a domain whose typical applications are developed employing the general reusable architecture produced. A Business Object (BO) is one of the main entities in a Business Model (as an example the Customer for the Sales domain, or the Supplier for the Purchases domain).

A specific Class Library (IBM Open Class Library in C++) has been employed, reusing its visual elementary controls (FrameWindow, EntryField, ListBox, Button, Container, Event, Event Handler.). Moreover a graphic interfaces visual builder (i.e. a visual tool that accept as input a visual description of a panel layout and give birth as output to the C++ classes that implement the panel itself) has been used for the production of the underlying code.

5.2 Domain characterization

The first activity has been the identification and the classification of the information concerning the context of the domain under consideration, specifically documents, manuals and literature that deal with graphic interfaces, settings of the type Model View Controller and the most widespread graphic and visual standards. Then requirements have been identified, analyzing the user interfaces of the applications modules that implement the typical insert, modify, cancel, query and list operations in different application domains. The common aspects characterizing these applications have been extracted, picking out the ‘look and feel’ that matches the user expectations. Requirements found have been formalized in a document named ‘Visual Standards’, documents that presents the structural and behavioral features of the graphical interfaces common to the applications. Moreover the document contains the description of the layout and of the event-driven actions of the main elements in the interface:

- Container Window (CW), i.e. the window that presents the details of the instances of a Business Object that comply with given filters;
- Setting Window (SW), i.e. the configuration panel for the customization of the Container Window;
- Object Edit Window (OEW), i.e. the window which guides the maintenance of a Business Object;
- Search Window (RW), which is a Container Window that displays the instances found with a query on the code or on the description;
- Key Window (KW), i.e. the exhibit of the identification of a Business Object.

5.3 Data analysis and modeling

In this step the requirements has been analyzed and object oriented models have been built, following the prescriptions of the UML methodology.

The first model, the Object Model has been designed to represent the object structure of the domain, considering:

- classes that implement main abstractions in the domain and the role of each of them in the system;
- relationships and associations among classes, together with their attributes (described by their type, cardinality, etc.);
- definition of suitable Message Trace Diagrams and scenarios to identify operations and attributes for each class.
The Object Model has been refined applying the concepts of generalization and aggregation, identifying commonalities and hierarchies among classes: at the end a synthetic object model, representing the common aspects of the system, has been produced.

A Dynamic Model accompanies the Object Model, presenting a dynamic description of the behavior of the system. The Dynamic Model is derived from the Message Trace Diagrams, describing the timing of the system, and from the State Diagrams, representing the state evolution of the system in response to external events.

The description of the variant aspects of the system complements these models. In particular, in the definition of the variants, we identify the kind of flexibility requested and the suitable variability class with the help of domain experts that know or that are able to forecast the possible variations that will be required to the applications in the domain. The descriptions of variants allow to reuse and to adapt an artifact in different applications and different contexts and their selections constitute an important step in the analysis process. The variation point have been collected and described in a classification document: the refinement of domain models into a set of cooperating frameworks has been based on this document.

An issue/decision document has got together technical and structural aspects and solution adopted during the development process, tracing the rationale behind them along the whole process and recording information that could not be completely clear or understandable to developers in next phases.

5.4 Definition of a reusable architecture

Taking as input the object oriented models produced in the previous phase, an analysis and refinement has been undertaken, considering at the beginning the classes. The classes identified in the models have been grouped into separate categories, basing on their relationships and interconnections. Classes having strong cohesion among themselves and presenting a scarce coupling with classes from other groups constitutes a good candidate framework.

Among the most important, the following categories have been extracted:

- **Container**: this module is in charge of the management of the container and search windows presenting Business Objects’ instances.
- **Settings**: this module defines customization settings for the container windows, both regarding visualization than information filtering and loading.
- **Controls**: this component performs the definition of the hierarchy of visual controls and of their event-handler.
- **Errors**: this module manages errors and visualization through a ListBox.
- **Window Manager**: it is in charge of handling OEW level events, activating standard actions scheduled.
- **Control Set**: this component realizes an interface layer between the User Interface and the Business Model.

Considering the variable aspects identified during the previous step, domain models have been modified in order to incorporate the required variation points, following the indications of some design patterns: ‘Template Method’, ‘Observer’, ‘Factory’.

The following frameworks are the result:

- **Display Framework**: this framework manages the window of the instances of a Business Object (in both the cases of a Container Window and a Search Window), supports the definition and activation of the Container Window through its settings, and control the uploading of the information from the database;
- **Action Framework**: this frameworks controls the creation of the Object Edit Windows and of the Key Windows, manages events both control events and Object Edit Window events, and show possible errors using a ListBox;
- **Control Framework**: this framework acts as an interface, uncoupling the User Interface from the Business Model, activates check operations on the values acquired (attributes from the User Interface) and transfer them to or from the Business Object.
The Control Framework is a white-box framework: it is used/reused specializing the Container Window class and introducing in a database table the attributes that can be shown for each Business Object. The Action Framework is a gray-box framework, considering that its use/reuse is obtained through specialization of the Window Manager class, reusing the visual components of the framework for building the Object Edit Window and the Key Window (that have to be linked to the Window Manager). The Interface Framework is a black-box framework: its parameters are described and bounded to specific instantiation values using a database table. The information concerning the whole architecture has been described in a diagram that summarizes the framework structure, along with the relationships and connections among them.

6. Conclusions

Domain analysis, even if introduced in the late 80s, has great potential still to exploit, in a way similar to software reuse which domain analysis fosters and supports. In this work it has been described an alternative domain analysis methodology, called Sherlock. This methodology has its roots in two domain analysis approach rather different: the PROTEUS methodology, which focus on domain variability and domain evolution and is based on object oriented techniques, and FODA, which uses a functional approach based on features (user-visible aspects of systems) and prescribes a rather complete documentation of the activities performed (introducing, for example, an issue/decision document). Sherlock is based on a reference incremental software process and uses object-oriented techniques; it focuses on variability identification and prescribes a complete documentation of the main activities and outcomes of the process. Sherlock fosters reuse practices through the production of reusable assets and frameworks and general domain architectures. The effects on reuse of Sherlock is not easy to understand, mainly because of the complexity of the environment and of the development process.

Sherlock has been experimented inside Thera S.p.A., an Italian firm producing software products, for the definition of a graphic interface module that is currently used in different applications. Sherlock has demonstrated in this occasion its potential effectiveness, even if a single experimentation can not be sufficient to assure an objective and valid results. To evaluate the impact of Sherlock we have defined a monitoring program based on the collection of different information, such as effort spent, product and process parameters (see [BSVV96] for an introduction to monitoring programs of reuse practices). A preliminar analysis of the data collected has shown some interesting aspects:

- The products delivered using Sherlock, during this initial phase, show an increased cost as compared with the expected one, but it has to be considered that the incremental costs associated with the application of Sherlock are concentrated in its first applications and will decrease in the next ones.
- The products delivered at the end of the project show an increased quality, in particular concerning modularity, documentation and maintainability, but seems to slightly lose execution efficiency ...
- Reuse seems to benefit from the adoption of Sherlock, both concerning quality and the cost of the process, but it has raised a learning problem: people has to be trained in the new technologies and need more time to understand and use the frameworks than usual small-scale components.
- The institutionalization of Sherlock and the related techniques (software architectures and frameworks) need to be formalized into the software process model and requires a start up time during which the people must learn how to effectively apply it; moreover the benefits will be achieved only in the mid or long term (similarly to reuse introduction) when more projects will use Sherlock and its results.

The current Sherlock approach is evolving, in particular some specific aspects are in expansion:

- It should be possible to extend the preliminary step regarding domain scoping with the adoption of a structured technique for collecting and formalizing system requirements and user needs; potential benefits are an early detection of commonalities and variations, together with improved description of possible evolution of the domain as gathered from the users' perception of the domain.
- The business analysis and the feasibility analysis present a great improvement potential: nowadays these activities are mainly performed basing on the expertise and capabilities people assigned to these tasks, but this practice can not give assurance of reliability and replicability, while on the verification and validation
side it is quite weak. Considering traditional economic techniques and market analysis methods, suitable guidelines and standardized practices should be devised aiming to formalize activities such as cost-benefits analysis, risk evaluation and technical feasibility evaluation, introducing the desired replicability and reliability characteristics.

- Currently variability of the domain is managed in Sherlock using plain variation documents: extending variability identification, classification and management through the use of a Configuration Management tool that organizes also versions and releases of reusable assets, will really improve the software process, allowing efficient packaging of new software system, considering needed variations and their specialization aiming to customize the final product aiming to meet the specific user needs.

References


[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.


Table of Contents

Special Issue on Parallel Object-Oriented Programming

Guest Editors: Boleslaw K. Szymanski, Charles D. Norton

M. Austern, R. Towle, A. Stepanov: Range Partition Adaptors: A Mechanism for Parallelizing STL

S. Baden: Software Infrastructure for Non-Uniform Scientific Computations on Parallel Processors

D. Gannon, S. Diwan, E. Johnson: HPC++ and the Europa Call Realization Model

Y. Ishikawa: MFC++ Approach to Parallel Computing Environment

A. Chien: ICC++ - A C++ Dialect for High Performance Parallel Computing

C. Kesselman: High Performance Parallel and Distributed Computation in Compositional C++

C. Norton, V. Decyk, B. Szymanski: On Parallel Object Oriented Programming in Fortran 90