Requirements Eng (2011) 16:117–132
DOI 10.1007/s00766-010-0113-9

ORIGINAL RESEARCH

Supporting agile software development through active documentation

Eran Rubin · Hillel Rubin

Received: 7 May 2010 / Accepted: 27 September 2010 / Published online: 13 October 2010
© Springer-Verlag London Limited 2010

Abstract Agile development processes are adaptive rather than predictive. Therefore, agile processes emphasize operational system code rather than its documentation. To overcome the absence of comprehensive documentation artifacts, agile methods require constant interaction between the system stakeholders. Ironically, however, some traditional documentation artifacts come to support this kind of interaction. In this study, we examine the relationship between software and documentation. We develop an approach that enables incorporating domain documentation to agile development, while keeping the processes adaptive. We also provide a system design that actively uses domain knowledge documentation. These ideas have been applied through the implementation and use of agile documentation support components.

Keywords Domain knowledge · Conceptual modeling · Requirements engineering · Agile documentation

1 Introduction

Agile development processes have become increasingly popular over the last several years. These processes attempt to enable more flexible and adaptive software than traditional software engineering processes do [22].

E. Rubin
Faculty of Technology Management,
Holon Institute of Technology, Holon, Israel

H. Rubin (✉)
Faculty of Engineering, Technion – Israel Institute of Technology, 32000 Haifa, Israel
e-mail: hrubin@technion.ac.il

Probably one of the main contributors to the success of agile methods is the dissatisfaction with the bureaucracy of traditional development methodologies. Agile methods require less documentation for tasks and promote implementation based on informal collaborations between system stakeholders [22]. While traditional software engineering methods emphasize careful planning and design, agile methods emphasize the actual software implementation.

However, this shift of emphasis is not without cost. Documentation is, among other things, used for knowledge sharing and reduces knowledge loss when team members become unavailable [3]. Since documentation is compromised when applying agile methods, important knowledge may be lost during and after system development.

Agile methods overcome documentation scarcity by significantly relying on constant collaboration between developers and users [3]. Relying on collaboration imposes a critical premise about the stakeholders involved—they must possess common knowledge and a common language to enable communication.

However, in many cases, building these common grounds may be very difficult to achieve. For example, in distributed development, where interaction is scarce and backgrounds are different.[1] Hence, without documentation, agile methods do not suggest ways for establishing the necessary infrastructure supporting collaboration.

It should be noted, however, that agile development methods do not preclude the use of documentation in their processes. Rather, in comparison with traditional software processes, agile development is less document-oriented.

[1] Therefore, organizations which apply distributed development practices tend to change agile methods to follow the distributed characteristic of their organization [7].

However, this is not a key point, but rather a consequence of the agile objective of being adaptive [51]. Therefore, in essence, it may be possible to support documentation in agile development methods without compromising the agile manifesto [60]. If documentation is adaptive and if the documentation supports people collaboration rather than replacing it, then documentation can be well aligned with agile development principles.

This research originates from the question concerning which kind of documentation can best support collaboration and how we can be able to integrate such documentation in agile development. That is, we examine whether there is a way to create an adaptive system for documenting the knowledge necessary for the interaction and collaboration between system stakeholders.

We propose agile documentation of domain knowledge gathered during systems analysis in traditional processes. In traditional processes, systems analysis is the stage in which the need for the system is established and system requirements are elicited. We examine traditional systems analysis since this is the stage in which all system stakeholders interact and a common understanding of the domain is established and documented. Therefore, within the process of systems analysis, we are able to find documents supporting collaboration, which are missing in agile development processes.

Once a set of collaboration supporting documents is identified, we establish a method to incorporate such documents in agile processes. Since agile development emphasizes the reference to working system code, we develop a way of having the identified documentation as part of the executable system code. More specifically, we suggest a system architectural design that enables adaptable documentation as part of the source code. We term our proposed system design Active Documentation Software Design (ADSD). Under this design, source code execution incorporates the execution of documentation statements, which in turn drive the processing of the system. Stated differently, with ADSD changes in the documentation change executable code and vice versa, changes in source code change the relevant documentation.

The concept of active documentation is based on the notion that implementation code is closely related to system design. These ideas are further developed in the sections that follow. Section 2 provides the background and motivation for this work. Section 3 elaborates on principles guiding the development of the architecture. Section 4 describes the architecture and the implementation of components for its support. A description of experience in using and applying the architecture is followed in Sect. 5. Finally, Sects. 6 and 7 incorporate the discussion and summary.

2 Background

2.1 Documentation in development processes

According to traditional approaches, software development processes should incorporate three iterative phases: Analysis, Design, and Implementation [29]. During this process, documentation artifacts are created. In fact, studies often refer to the entire development process as a document creation process [40, 41, 63]. The different documents include requirements, interviews, prototypes, and more. A significant set of documentation artifacts are *models*. Models serve as a bridge between the analysis and the design phases [51]. They are conceived during the analysis phase and assume different transformations throughout the design phase.

Among the development phases, the design phase is probably the most loosely defined and is considered more creative than a mechanistic process [61]. This phase consists of progressive decompositions toward accumulating details [59]. Each decomposition results in a new design document. It has often been claimed that the design documents created during this phase fail to be synchronized with reality as the system evolves [9, 15, 41]. Since system maintenance and update constitute the majority of a development lifecycle [15], this situation accounts for much of the criticism on the traditional development processes. The processes were too cumbersome to support limited benefits [22].

Agile development methodologies emerged as a result of the difficulties described in the preceding paragraph. Agile methods embrace the unique attributes of working code. Working code tells the stakeholders what they really have in front of them, as opposed to models with promises which state what they *will* have in front of them [33]. In agile processes, a minimum set of principles is provided rather than detailed development rules. Hence, agile methods aim at overcoming the deficiencies in the traditionally ambiguous design phase by avoiding intermediary artifacts.

However, as noted, while it may be that traditional development methods over-document, a credible weakness of agile methods is of insufficient documentation [51]. In both agile and traditional approaches, during the processes known as requirements engineering, the development team must acquire the same information and the customer must make the same decisions about the system [38]. While theoretically agile processes need to use the same kind of knowledge to facilitate an effective development process, they cannot reconcile its documentation with the agile principles. Therefore, due to the difference in development practices, traditional development usually embodies the requirements in a written document, while agile development does not [38].

2.2 Requirements engineering in agile processes

The major difference between agile and traditional approaches is not *whether* to engage in requirements engineering (RE), but rather *when* to do so [38]. In agile development, requirements are built throughout the development process. Therefore, there are no models documenting acquired domain knowledge. Rather, knowledge is only found in code, test cases, and programmers' memory [49]. The way RE is supported in such an environment is implied in the practices taken to facilitate source code modification. In this respect, agile RE is supported by following the following principles:

Domain Knowledge should be constantly available for easy reference—This principle is reflected in the "Small Releases and On-Site Customer" practice in Extreme Programming (XP), which states that users or domain experts work directly with the programmers, answering questions about details and resolving misunderstandings as they come to light [38]. Similarly, Crystal and Adaptive Software Development (ASD) advocate an end-of-iteration process and team reviews. ASD and Scrum use end-of-iteration reviews with customer focus groups [33].

Attempt to validate requirements explicitly in some source code—This principle is reflected in the "Test First" practice, which states that programmers are required to write tests for each bit of code before writing the code. Thus, the software and a refined test suite grow together [38].

Attempt to get code associated with requirements easy to locate, understand, and update in source code—This principle is reflected in the "Refactoring" principle. Refactoring is the act of changing working code so it still carries out the same tasks but in a way that is easier to understand and modify [38].

The way updated requirements are embedded in code should be shared between developers—This principle is reflected in the "Pair Programming" practice in XP, which sets that all production code is written by two programmers together. Similarly, Crystal, Scrum, and ASD advocate close collaboration practices including barrier-free collocated teams. Lean Development stresses team interaction [33].

Development should follow users' view rather than system view—This principle is reflected in the practice of using "story cards" in XP. A similar notion is found in Scrum, using the term "backlog". ASD and Feature-Driven Development refer to features. Basically, agile approaches plan features, not tasks, because features are understood by customers [33].

Hence, different practices are taken in agile approaches to facilitate requirements engineering. While these practices are different from traditional practices, they help elicit similar knowledge. The key difference between agile approaches and traditional approaches, however, is that while traditional approaches explicitly document the gathered knowledge, agile approaches tend to refer to source code as the only documenting artifact.

2.3 Implementation as a design document

Advocates of agile development processes posit that system source code should serve as the "true design document" of the system [55]. In this section, we suggest that while source code can indeed serve to reflect many aspects of the system design, it is not a true replacement of the traditional design documents. We begin by reviewing the forces that lead to this conception in agile methods.

Traditionally, the design process gradually moves from the "problem space" to the "solution space" [10]. However, as different paradigms of programming emerged, the difference between the problem and solution domain became less and less distinct [29]. Initially, programs did not exhibit problem-level information that could be understood or modified. The programmer was not required to make a program understandable and modifiable, but rather programs were merely measured by their efficiency and whether they could accurately solve a specific problem. Presently, software engineering emphasizes that programs should be written to be understandable and maintainable. An important step in this direction was the introduction of Object-Oriented Programming (OOP) paradigm, in which programmers are thinking more in terms of the problem domain [1].

The more source code related to the problem space, the more difficult it was to distinguish between design and implementation. For example, Heramann [32] states that some studies [16, 35] have pointed out that object-oriented analysis refers only to domain objects, while object-oriented design refers also to future system objects. He also mentions that other studies have not yet clearly distinguished between the designs [57, 58].

This state of affairs has led to the perception suggesting that source code is in fact the design document. While such arguments have merits, we cannot claim that source code and other design models created during the development process are equivalent. Two major differences exist between these two concepts:

1. Source code is less accessible to different stakeholders in the organization than design documents are.

Basically, only a professional programmer can understand the source code. Further, familiarity with the development environment is needed in order to navigate through the source code.

2. While there may be an overlap of information available in source code and other design documents, there are still many differences. For example, source code contains more information since it holds implementation-related details. Such details include networking details and security issues. On the other hand, in other respects, source code contains less information due to differences in its semantic power. For example, most programming languages do not support the explicit representation of roles, while roles are often found in early design documents.

In short, traditional design documents and system source code are highly related. Some of the knowledge found in traditional design documents can also be found in source code. However, fundamental properties associated with the structure and composition of code to date distinguish it from design documents.

3 Supporting agile documentation

3.1 Documentation objectives

Important documents, which are absent when using agile methods, are those created during the analysis phase in traditional development processes.

Systems analysis was defined as the phase in which reasons for the need of an information system are understood [19]. However, realizing that inaccurate requirements account for many problems encountered during software development [e.g. 5, 48], the concept of systems analysis has undergone some transformations. Gradually, reference to the systems analysis phase has shifted from "Systems Analysis" to "System Requirements Analysis" [e.g. 24], "Requirements Analysis" [e.g. 28], and most commonly today to "Requirements Engineering" (RE).

RE is concerned with interpreting and understanding stakeholders' terminology, viewpoints, and goals [50]. Therefore, models support all RE activities [42]. More specifically, in traditional development methods, RE processes incorporate the modeling of the existing domain as well as alternative hypothetical systems [42]. In the literature, these two types of models are often referred to as "conceptual models" and "system models," respectively.

Conceptual models represent the problem understanding [20] and facilitate understanding among various stakeholders in the organization [47]. Mylopoulos [46] describes conceptual modeling as the "activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication" [46]. Both conceptual models and system models are essential for the RE process—conceptual models for understanding the problem domain and system models for generating requirements associated with the system.

Since the understanding of the application domain and alternative systems changes as development process propagates, both conceptual and system views are dynamic and evolving. In this respect, evolved system code naturally represents the most accurate *system* model. Therefore, as the agile movement has pointed out, the need for the original *system* models may be challenged. Nonetheless, this is not the case for *conceptual* models. As the understanding of the domain evolves, *conceptual* models are not accurately represented anywhere. Ironically, while conceptual models are lost in agile development methods, they could be highly relevant when using such methods. Agile development requires constant and efficient communication between all system stakeholders, which is the core purpose of conceptual models. Since our objective is to facilitate their documentation in agile processes, we examine the content of conceptual models.

3.2 Documentation constructs

In order to enable agile documentation of conceptual knowledge, we examine the core documentation constructs used when creating conceptual models under the traditional RE processes.

As stated by Nuseibeh and Easterbrook [50], to a great extent, the elicitation technique used during RE is driven by the choice of the modeling scheme, and vice versa. To this end, Nuseibeh and Easterbrook [50] mention four general categories of modeling approaches used in RE: Data Modeling, Behavioral Modeling, Enterprise Modeling, and Domain Modeling.

Over the years, an extensive number of modeling languages with varying levels of complexity have been developed under each modeling approach. Naturally, as a modeling construct grows in complexity, it becomes more difficult to use, and its appeal for agile documentation is reduced. However, in each of the four modeling approaches, there seems to be a set of core modeling constructs, which are commonly used for communication about the domain. By enabling the use of these constructs in working implementation code, we wish to enable active documentation of the domain. Therefore, in the following subsections, we review the core constructs used under each modeling approach.

It is important to note that the approaches are not mutually exclusive. Rather, each approach takes a different view as to the aspects that should be emphasized to facilitate requirements elicitation. Data modeling approaches deal with understanding the information that needs to be stored, maintained, and managed by the information system. Behavioral modeling approaches deal with the dynamic or functional behavior of stakeholders and systems. Enterprise modeling approaches deal with understanding an organization's structure, the business rules that affect its operation, and the goals, tasks and responsibilities of its members. Finally, domain modeling approaches deal with understanding the impact of the system on the domain and the way it helps to control the domain.

Since the approaches are not mutually exclusive, the meaning of some of the core constructs used in different approaches may overlap with each other. Hence, to facilitate clear documentation of domain knowledge in Information Systems (IS) code, following our review, we compare the constructs and integrate them. This enables us to provide a set of constructs that do not overlap with one another, are well defined, and that their inter-relationships are clearly understood.

3.2.1 Data modeling

Data modeling aims at helping stakeholders make decisions about the type of information the developed system should represent. These models help communicate the correspondence between the information system and the real world [50].

Some of the better known data-based approaches are based on Entity-Relationship (ER) modeling [12]. ER diagrams consist of entities, relationships, and in the extension model termed E-R-A, also attributes. The ER model has evolved over time, and today the Extended Entity Relationship model (EER) is often referred to in conceptual modeling.

There are many other data-based modeling methods adopting similar constructs identified in the ER-based approaches. For example, the Object-Role-Modeling (ORM) approach [25]. ORM views the world in terms of things that play roles. In essence, in ORM, the way an entity participates in a relationship is its role. Relationships can be of various arties, which means that a relationship can be defined to include virtually as many roles as desired. In this respect, similar to the EER approach, ORM also defines cardinality constraints on relationships. That is, they provide a way to state that a relationship is restricted on the number of participating entities.

The ER-based approaches have gradually evolved, and object-oriented approaches can be considered as their consequents. Object-oriented techniques include objects, attributes, and structural relationships – generalization and composition. A significant difference from the EER approach is that in the object-oriented paradigm, objects additionally incorporate services/methods. Accordingly, the core domain aspects captured in the data modeling approaches are the following:

- *Entity*—a "thing" in the domain that can be distinctly identified.
- *Operations*—services or tasks a thing can execute or perform.
- *Entity sets/entity types*—a group of entities of the same type.
- *Relationship*—a relation between entities.
- *Relationship sets*—a relation between entity sets.
- *Value sets*—a range of values.
- *Attributes*—mappings from entity sets or relationship sets to value sets.
- *Constraints*—relationship sets and attributes can have cardinality constraints

3.2.2 Behavioral modeling

Behavioral modeling approaches emphasize the system behavior rather than the domain. Still, in the behavioral perspective, the attempt is to identify the events that occur in the real world, the information affected by their occurrence, and the functions that are invoked to cause this effect.

In general, Rolland and Prakash [56] identify three categories of behavior models:

1. Control flow is the oldest and most popular form of behavior model. Control flow iterates through steps following the completion of a previous step, regardless of the availability of inputs.
2. Data flow models iterate through steps upon input provision. In this case, inputs are provided directly from outputs of other steps.
3. State machines iterate through states upon events in the environment. The inputs to a step are calculated as part of the step itself. State machines comprise the basis of UML's behavioral models.

Underlying all the state-oriented techniques is the finite state machine. A finite state machine is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an event, the machine operates and generates state changes. Hence, the core domain aspects captured in the behavioral modeling approaches are those defined for state machine that can relate to the domain. Accordingly, the core behavioral domain aspects are as follows:

- *States*—represent the state of the domain.
- *Transition*—a state change from the source state to a destination state

- *Pre-conditions/post-conditions*—define what causes a transition and what the consequence of a transition is
- *Activities*—performed as long as the state is active.
- *Events*—trigger transitions between states

3.2.3 Goal-oriented\business enterprise modeling approaches

Enterprise modeling concerns the business along with its goals. Central to this modeling approach are actors and goals. An actor is an active entity that carries out actions to achieve goals [37]. In turn, different definitions are available for the term *goal*. However, in general, all definitions reflect the idea that a goal is a state that is desired to achieve [43] or a state of affairs in the world that the stakeholders would like to achieve [44].

A famous example of enterprise modeling is the i* model [64]. In i*, the participants of the organizational setting are actors with goals. These actors depend on each other in order to fulfill their objectives and perform their tasks.

Another popular goal-oriented approach is KAOS [18]. The main emphasis of KAOS is on the formal proof that the requirements defined for the envisioned system match the goals.

GBRAM [6] is another popular method that uses goals as the means to elaborate and structure system requirements. Similar to the other goal-oriented approaches, a system and its environment are represented in GBRAM as a collection of agents, which are defined as entities or processes that seek to achieve goals within an organization or a system.

With the notion of goals and actors, the different goal-oriented techniques model stakeholders' interests and how they might be addressed by the various configurations of systems and environments.

Relationships between goals are aimed at capturing situations in which goals positively or negatively support other goals [43]. For example, a conflict relation between two goals is introduced when the satisfaction of one of them may prevent the other from being satisfied [43]. In the process of goal operationalization, new and less abstract goals related to original ones through causal relations are identified [45]. In this process, goals are essentially decomposed to *and-refinements* and *or-refinements.* In *and-refinements*, a set of subgoals sufficient for satisfying the parent goal are identified. In *or-refinements*, different ways for achieving a parent goal are identified.

Dependency is another common construct in goal-oriented approaches. In i*, dependency is a relationship in which one actor (the depender) depends on another actor (the dependee) for something (the dependum) to be achieved [65]. Three types of dependencies can be distinguished: goal dependency, resource dependency, and task dependency. In a goal dependency, an actor depends on another actor to fulfill a goal. In a resource dependency, an actor depends on another actor to provide a resource, which is a passive entity [36]. In a task dependency, an actor depends on another actor to carry out a task, which is a particular way of doing something [44].

Similar notions are used in other goal-oriented approaches and some also repeat the constructs used in data and behavioral-oriented approaches. For example, KAOS goal definition patterns are used for specification of goals. These patterns include *achieve, cease, maintain, optimize*, and *avoid* [14]. KAOS objects are very similar to those discussed in data model approaches, and objects can be entities, relationships, or events. Finally, KAOS operations are input–output relations over objects and relate to behavioral modeling constructs. In essence, operations define state transitions and have pre-, post-, and trigger conditions.

Accordingly, the additional domain aspects captured in goal-oriented modeling approaches are the following:

- *Goal*—a state that actors want to bring about.
- *Goal relations*—the relation between goals capturing where goals positively or negatively support other goals.
- *Actor*—an active entity that carries out actions to achieve goals.
- *Resource*—a passive entity.
- *Task*—specifies a particular way of doing something.
- *Dependency*—a relationship in which one actor depends on another actor.

3.2.4 The domain knowledge approach in requirements engineering

Domain modeling approaches highlight the need for a clear relationship between requirements and specifications [68]. The view under this approach is that specifications together with relevant domain knowledge should be sufficient to guarantee that requirements are satisfied. This is formalized in Zave and Jackson [68]:

$$K, S| - R$$

where K is knowledge about the problem domain, S is the specification of the solution, and R is the problem requirement.

Viewing RE under the domain knowledge approach dates back to the work of Dubois et al. [21], who suggested the Entity-Relationship-Attribute-Event, (ERAE) language. This language borrows from ideas in Semantic Networks and Logic. More recently, Problem Frames [26, 34] reflect this point of view.

The uniqueness of the approach is in the clear attempt to distinguish between the system and the environment. By emphasizing problems rather than solutions, one can exploit the understanding of a problem class [26].

Within a problem frame, a machine domain is defined and modeled. The machine domain is the system built together with its underlying hardware. In contrast to the machine domain, other domains, termed given domains, represent parts of the world that are relevant to the problem. These domains include physical events and state that are causally related. The different domains may share events and state information. These are termed shared phenomena. Phenomena shared between two domains are observable by both, but controlled only by one of them.

In the problem frames framework, a requirement is defined as a condition in the problem domain that the machine domain must guarantee to qualify as a solution to the problem. With this, the domain constructs of the domain modeling approaches are [13, 34]:

- *Phenomenon*—an element of what we can observe in the world.
- *(Given) domain*—is a set of related phenomena that are usefully treated as a behavioral unit for some purpose.
- *Event*—an occurrence at some point in time, regarded as atomic and instantaneous.
- *State*—a relationship between two or more individuals that can be true at one time and false at another.
- *Value*—an individual that cannot undergo change over time. It is a kind of a phenomenon. The values in which we are interested are things represented by numbers and characters.

3.2.5 Integration

The preceding sections revealed that the four RE approaches emphasize modeling different parts of the domain. Therefore, each approach takes a different set of modeling constructs. However, the different approaches are not mutually exclusive, and the knowledge they aim at documenting may overlap. Consequentially, the constructs used to model the domain in the different approaches may have similar meanings.

Under these circumstances, the challenge is to provide a coherent and well-defined set of constructs that will enable the representation of the domain knowledge gathered in any of the four RE approaches. We approach this task by referring to the Enterprise Ontology [62].

Generally, an Enterprise Ontology defines a set of constructs and their interrelations, which together are used to explicate the business domain. Different Enterprise Ontologies and supporting frameworks have been proposed over the years [e.g., 23, 62, 67] designed to support different activities. Uschold et al.'s Enterprise Ontology is semi-formal. That is, it provides a glossary of terms expressed in a restricted and structured form of natural language. Hence, it provides the rigor required for the definition of constructs, while keeping the natural expressiveness required to support communication.

In the Enterprise Ontology, Uschold et al. [62] define a set of constructs that enable representing any phenomenon at the business domain. As such, their constructs are well defined and the interrelations between them are made clear. By corresponding between the constructs used in the four RE approaches and those of the Enterprise Ontology, we are able to distinguish between the core constructs, as well as identify their compositions and interrelations.

However, not all constructs used in the Enterprise Ontology are necessarily relevant for representing RE-based domain knowledge (REDK). The REDK representation constructs are those naturally used to discuss and identify system requirements. The constructs of the Enterprise Ontology, on the other hand, come to represent anything in the domain, including elements that may not be captured in RE. Furthermore, it is not clear how the constructs identified in the previous section relate to Enterprise Ontology constructs. There is a need to identify which constructs in the Enterprise Ontology can be used to represent RE domain knowledge.

Hence, in this section, we identify a subset of the constructs defined in the Enterprise Ontology, namely the constructs that can be used to represent REDK. For this purpose, we apply a mapping from the constructs identified in the four RE approaches to constructs defined in the Enterprise Ontology. This process yields a mapping to the following Enterprise Ontology constructs. (The mapping itself is given in Table 1.) The explanations taken from the Enterprise Ontology for these constructs are as follows:

- *Entity* A fundamental thing in the modeled domain.
- *Relationship* The way that two or more entities can be associated with each other. Within a relationship, an entity may have a role (e.g. a person may be a customer in a sale). Alternatively, an entity may be seen as an attribute of another entity (e.g. date of birth of a person).
- *Roles* An entity may have a role reflecting its relationship with other entities. Some relationships are special in that they entail some notion of doing or cognition. We refer to an entity involved in such relationships as an *actor*. Relationships among actors may entail some view (activity or cognition) for one of them. This view indicates the actor has an *actor role*.
- *Actors* Certain roles in a relationship are special in that the playing of these roles entails doing or cognition. These are called actor roles. Entities playing such roles are called actors. Hence, if the role played by an entity

Table 1 Requirements engineering domain knowledge constructs mapping

Concept	Goal Oriented	Data based	Behavioral	Domain Based
Entity		Entity (e.g. [12])		Phenomenon (e.g. [26])
Relationship		Relationship (e.g. [12])		
Actor	Actor (e.g. [6]) Agent (e.g. [11])	Entity (e.g. [57])	–	Part of Phenomena (e.g. [34])
Role (Actor Role)	Role (e.g. [66])	Relation (e.g. [12]) Role (e.g. [25])	Operations	–
Goal/Subgoals (Purpose)	Achieve goal/maintain goal (e.g. [18])	–	–	
Help Achieve	Subgoals (e.g. [18]) Goal dependency (e.g. [66])			
Service (Activity)	High-level task (e.g. [66])	Operations/methods (e.g. [16])	Activities	–
Resource	Resource (e.g. [66])	Entity (e.g. [12])	–	–
Constraint (Activity Specification)	Cease goal; avoid goal (e.g. [18])	Constraints (e.g. [12])	–	–
State and attributes (State of Affairs)	–	Value (is part of attributes) (e.g. [12])	State (e.g. [53])	State (e.g. [68])
Pre-Condition	Pre-condition is part of a task and a resource dependency (e.g. [66])		Events (e.g. [27]) Pre-condition is part of transition (e.g. [54])	Events (e.g. [26])
Concept	Goal oriented	Data based	Behavioral	Domain based
Effect	Effect is part of a task and resource dependency (e.g. [66])		Post-conditions (e.g. [27]) Effect is part of transition (e.g. [54])	

entails some notion of doing (such as exposing services), or cognition (such as desiring a goal), the entity is an actor. Actors are active entities in the organization. An actor can either be a person, an organizational unit, or a machine that performs some activity.

- *Resources* When an entity does not expose a notion of doing or cognition in any relationship, it is a resource. A resource is the role of an entity in a relationship with an activity whereby the entity is or can be used or consumed during the performance of the activity.
- *Activities/services* Something done over a particular time interval. An activity has pre-conditions and effects, is performed by one or more doers, can be decomposed into more detailed subactivities, may entail the use or consumption of resources, can have authority requirements, and may have an owner.
- *State* (*of affairs*) A state is some kind of situation that can be thought of as holding or being true. An attribute is a relationship between two entities (referred to as the 'attributed' and 'value' entities) where for any particular attributed entity, the relationship may exist with only one value entity. A state of affairs is a situation characterized by any combination of entities that are in any number of relationships with one another.
- *Purpose/goals* A goal is the role of a state whereby the actor wants, intends, or is responsible for the full or partial achievement of the state. A goal (achieve) is the realization of a state (of affairs).
- *Help achieve* A relationship between two states of affairs whereby one state of affairs contributes to or facilitates the achievement of the other state of affairs. The help achieve relationship is particularly important when the states of affairs are goals. In this case, the help achieve relationship may define a directed acyclic network of goals, which gives rise to a notion of higher- and lower-level Purposes/Goals.
- *Constraints* The Enterprise Ontology mentions three types of constraints: 1) Effect—a state that is brought about by an activity. 2) Pre-condition—a state required

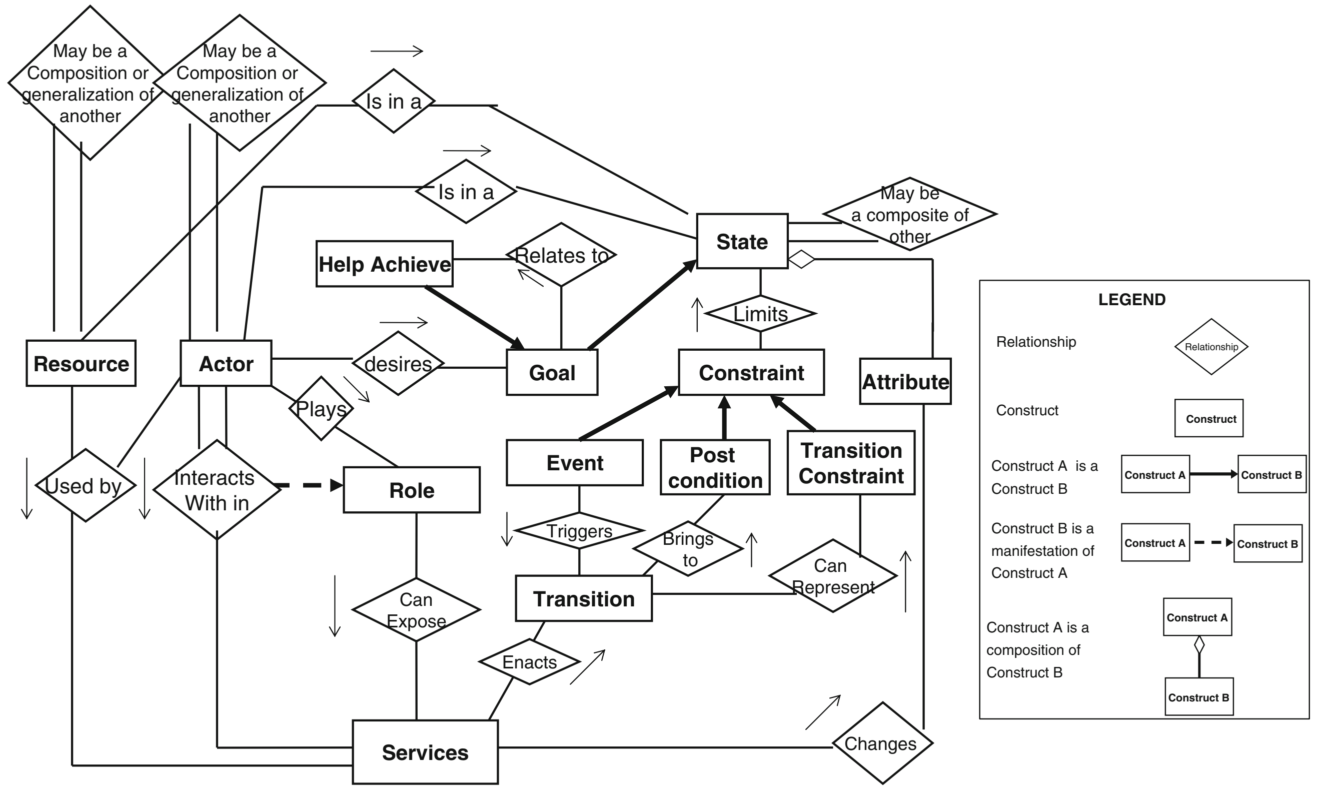

Fig. 1 Metamodel of requirements engineering domain knowledge constructs

to be true in order for the activity to be performed. 3) Restrictions on the range of activities in the universe.

- *Pre-condition* A state of affairs required to be true in order for the activity to be performed.
- *Effect* A state of affairs that is brought about by an activity.

In Table 1, which depicts the mapping, the leftmost column incorporates the Enterprise Ontology construct.

Based on this mapping, it can be seen that the Entity and Relationship constructs are more abstract, and under the REDK constructs domain, other constructs subsume them. Entities are subsumed by resources and actors. Relationships are subsumed by roles and attributes. We also note that pre-conditions and effects are often used together both in RE literature and in the Enterprise Ontology. Specifically, the two together are part of the definition of a service in the Enterprise Ontology and relate to transitions and dependencies in the RE approach. Therefore, we compose them under one construct – *Transition*.

Hence, based on the RE literature and the Enterprise Ontology, we have identified eight fundamental domain-related constructs: *Actors, Roles, Resources, Services, Goals (that relate to Help Achieves), Constraints, Transitions (Incorporating Pre-conditions and Effects),* and *States (Consisting of Attributes).* The relations between the core constructs are depicted in Fig. 1.

4 Representation of domain knowledge in the system code

Following our identification of the core constructs used for RE domain knowledge representation, we turn to the task of enabling agile use of those constructs. That is, we suggest an infrastructure in which the identified REDK constructs can be used by developers to represent domain knowledge in system code.

We propose accomplishing this by enhancing the traditional object-oriented programming paradigm, namely enabling using REDK constructs in object oriented code. This in turn, should enable system developers to directly document in code things such as specific actors, resources, roles, goals, services, and constraints from the organization's domain. Hence, using REDK constructs in system code enables explicit documentation of domain knowledge as part of the working code.

Therefore, we provide a new class base. Our class base to support the use of REDK constructs is depicted using UML notation at the lower part of Fig. 2. These classes are related, by a one-to-one relationship, to the REDK representation constructs of Table 1. That is, each construct of Table 1 has a matching class in the figure. Therefore, all constructs can be used to document domain knowledge. We see that similarly to the interrelations

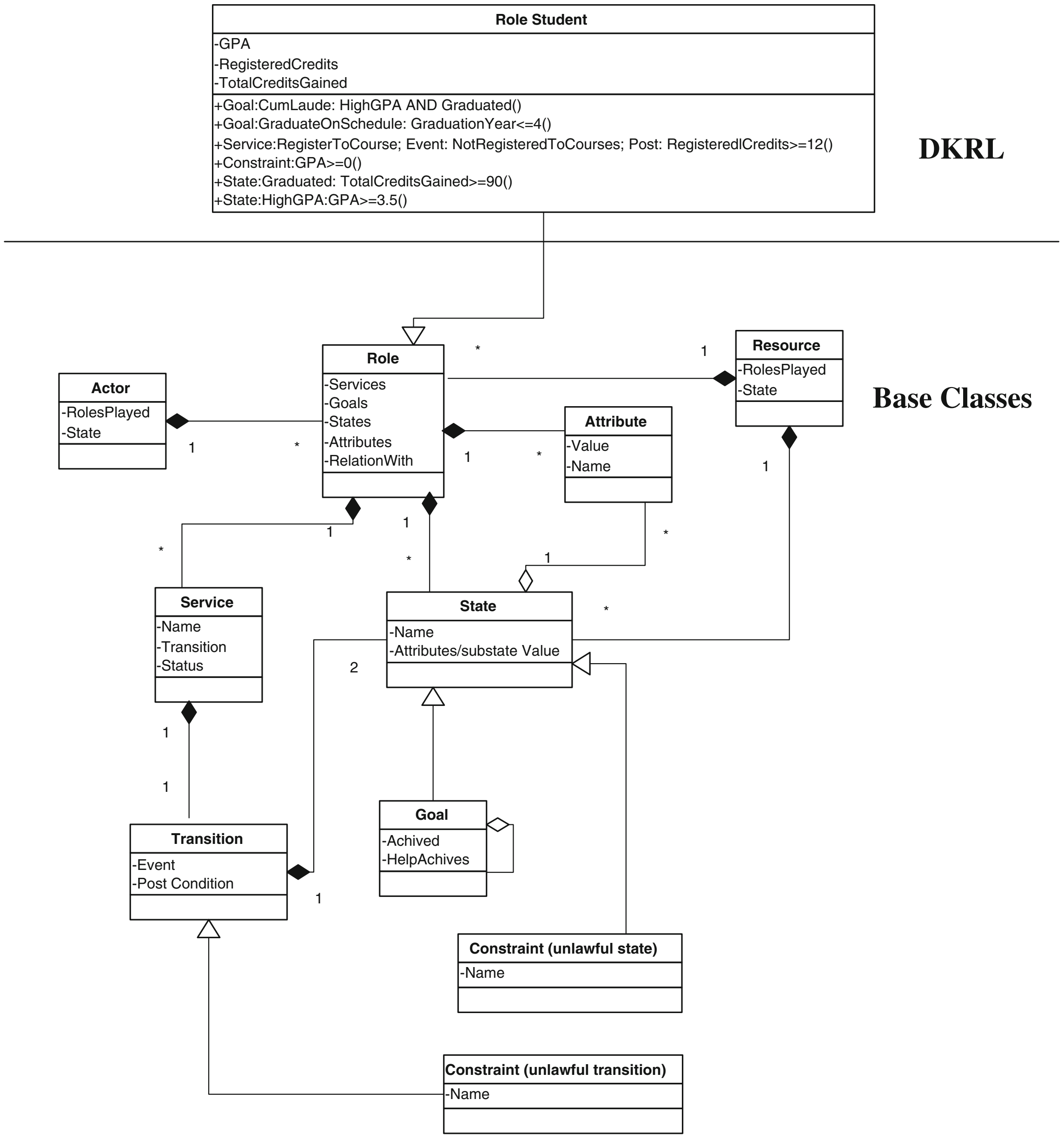
Role Student
-GPA
-RegisteredCredits
-TotalCreditsGained
+Goal:CumLaude: HighGPA AND Graduated()
+Goal:GraduateOnSchedule: GraduationYear<=4()
+Service:RegisterToCourse; Event: NotRegisteredToCourses; Post: RegisteredlCredits>=12()
+Constraint:GPA>=0()
+State:Graduated: TotalCreditsGained>=90()
+State:HighGPA:GPA>=3.5()
DKRL
Role
-Services
-Goals
-States
-Attributes
-RelationWith
Resource
-RolesPlayed
-State
Base Classes
Actor
-RolesPlayed
-State
Attribute
-Value
-Name
Service
-Name
-Transition
-Status
State
-Name
-Attributes/substate Value
Goal
-Achived
-HelpAchives
Transition
-Event
-Post Condition
Constraint (unlawful state)
-Name
Constraint (unlawful transition)
-Name

Fig. 2 Conceptual subsystem base classes used when representing knowledge at DKRL

between REDK constructs reflected in Fig. 1, each class can involve using other classes, therefore the association between classes at the bottom part of the figure. For example, since a role may include a service, we see the association between the role and the service classes in the figure.

The top part of Fig. 2 illustrates the use of our base classes to represent specific domain knowledge. The class above the horizontal line reflects specific domain knowledge linked by an "is-instance-of" relationship to the REDK role construct at the lower part of the figure. We have termed the area above the horizontal line Domain Knowledge Representation Layer (DKRL). Specifically, Fig. 2 shows that the meaning of the role *Student* in the domain is made explicit at the DKRL.

We note that knowledge represented using the REDK constructs is usually incorporated in the information system code. However, without explicit representation constructs, this knowledge is eventually intertwined with other implementation elements. For example, under the Object-Oriented (OO) paradigm, actors, resources and roles, which are associated with the domain, may often have representation using the same construct (i.e. the class construct) in the final implementation code. Further, elements that have no domain meaning may be associated with domain-related classes in the implementation code. For example, while a *student* in the domain typically does not incorporate any file handling services, the final implementation class may very well incorporate such services.

Therefore, in our approach, we provide more than just means for using REDK representation constructs. We also provide a system design facilitating the separation of the represented REDK (i.e. the DKRL) from other types of knowledge in the IS code. Hence, using REDK constructs under the suggested system design comes to facilitate the creation of domain models that reside at the DKRL. These domain models are part of the executable code, require minimal programming knowledge to understand, and are accessible to all system stakeholders.

5 The system design

We provide a system architectural design with a supporting mechanism that enables a physically isolated conceptual model to be used by other remote system components. More specifically, we suggest an architectural design in which the domain knowledge is separately located at what we term the *Conceptual Subsystem*. This domain knowledge is used by other system components, which together we term the *Processing Subsystem*.

Generally, under our proposed design, we facilitate a two-way interaction channel between the processing subsystem and the conceptual subsystem. During system runtime, changing the state of an element related to documented domain knowledge is done through the conceptual subsystem. That is, the processing subsystem updates states related to the domain through the conceptual subsystem. Conversely, based on documented domain knowledge, the conceptual subsystem updates the processing subsystem once processing needs to be done.

We have implemented system components to support such an interaction and have used them on a case study of a student registration system, adapted from Barker and Palmer [8]. We illustrate the workings of our proposed system design based on this experience.

Briefly, the considered system handles tasks associated with student enrollment services. For example, the system enables enrollment into courses, a process students have to undertake if they are not registered to any course. During the student's enrollment process, the system enables students to keep track of their grade point average (GPA), which helps them consider enrolling to particular courses. Students typically consider enrolling to courses which will enable them to complete their degree in time and improve their chances to complete it with honors.

Notably, being able to define the most basic system functionalities above entails that considerable domain knowledge was accumulated:

1. Some of the people in the domain play the role of students.
2. In the domain, students have a GPA, a graduation year, and a total number of credits of courses in which they are currently enrolled.
3. In the domain, students desire to complete their degree cum laude.
4. In the domain, students desire to complete their degree within the timeframe of no more than 4 years.
5. In the domain, completion of degree cum laude requires a GPA greater than or equal to 3.5.
6. If someone in the domain is a student, he/she must always be registered to 12 points worth of credit courses. If this is not the case, the system needs to start the course registration processes.

Under our proposed design, all elements of the domain knowledge are explicitly documented at the Conceptual Subsystem. Figure 3 illustrates how this is done by using the REDK representation constructs.

Using the role representation construct, it is made clear that the student role entails having a GPA, a graduation year, and a total number of credits as the state-composing attributes. Further, using the goal construct, the two goals of students are documented. Next, using the service construct, the pre-conditions and post-conditions of the course registration service are documented. Further, constraints are documented.

With domain knowledge documented at the DKRL, we now turn to describe how we support the interaction between the conceptual subsystem and the processing subsystem. In order to relate to the domain knowledge documented at the conceptual subsystem, the processing subsystem creates instances of the documented knowledge during runtime. In our example, the processing subsystem will create instances of actors and the student role which they play. These instances reside at the conceptual subsystem, and their creation automatically derives the creation of other relevant instances; such as instances of the documented goals, services, and constraints of the student role. We term the area in which the instances reside the *Instance Base*. In Fig. 3, some of the created instances are depicted. These instances are shaded and can be found at the lower right part of the figure.

Whenever the processing subsystem needs to update states related to the created instances, the updates are done at the conceptual system. In our example, updates of a student's GPA or the number of credits in which a student is enrolled are done at the conceptual subsystem on the respective instance of the student role.

Once such updates are made at the conceptual subsystem, the conceptual subsystem examines relevant instances at the Instance Base. In this process, the conceptual subsystem

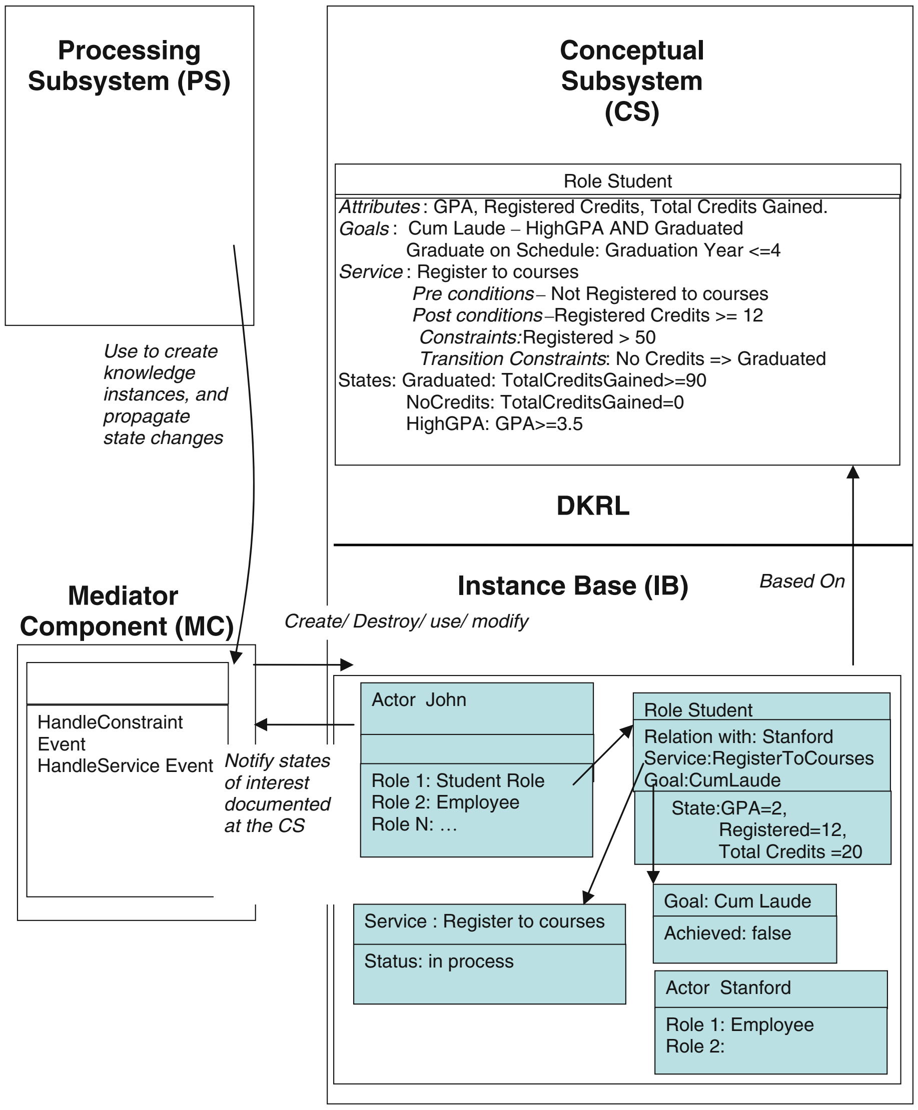
Processing Subsystem (PS)
Conceptual Subsystem (CS)
Role Student
Attributes: GPA, Registered Credits, Total Credits Gained.
Goals: Cum Laude – HighGPA AND Graduated
Graduate on Schedule: Graduation Year <=4
Service: Register to courses
Pre conditions– Not Registered to courses
Post conditions–Registered Credits >= 12
Constraints:Registered > 50
Transition Constraints: No Credits => Graduated
States: Graduated: TotalCreditsGained>=90
NoCredits: TotalCreditsGained=0
HighGPA: GPA>=3.5
Use to create knowledge instances, and propagate state changes
DKRL
Instance Base (IB)
Based On
Mediator Component (MC)
Create/ Destroy/ use/ modify
HandleConstraint Event
HandleService Event
Notify states of interest documented at the CS
Actor John
Role 1: Student Role
Role 2: Employee
Role N: …
Role Student
Relation with: Stanford
Service:RegisterToCourses
Goal:CumLaude
State:GPA=2, Registered=12, Total Credits =20
Goal: Cum Laude
Achieved: false
Service : Register to courses
Status: in process
Actor Stanford
Role 1: Employee
Role 2:

Fig. 3 System design and an example instance of use

identifies whether constraints have just been violated, whether goals have just been fulfilled, and whether services need to be performed.

In our example, once the GPA value changes, the system identifies whether the *HighGPA* state has been reached. Similarly, once the number of credits enrolled in changes, the conceptual subsystem identifies whether the *Not Registered to courses* state has become true. Arrival at one of these states will suggest that a service may needs to start, a constraint may have been violated, or a possible goal has been met. If indeed one of the latter occurs, the processing subsystem is notified via a complementary notification event. In our example, changing the GPA to an average higher than 3.5 may entail being *CumLaude*, if the student has also graduated. In such a case, the conceptual subsystem sends a notification event to the processing subsystem.

Under this framework, based on the documented domain knowledge, the processing subsystem is notified once an instance reaches a domain-based situation of interest.

At the processing subsystem, different implementations can be defined for the different events. For example, in a case of a violated constraint, an error message may be generated by the processing subsystem. Alternatively, when the 'event' state of a documented service is reached, the processing subsystem will execute the methods that should bring the object to service's documented post-condition.

In our example, if a student is found to become *not registered to courses,* the notification event sent from the conceptual subsystem to the processing subsystem will trigger a registration process at the processing subsystem. This process will enable the student to use the system to register to new courses. Consequentially, during process execution, once the student's number of registered credits will reach a total greater than 12, the conceptual subsystem will identify that the post-condition state has been arrived at. In turn, identifying the arrival at the post-condition state triggers a new notification event to the processing subsystem; i.e. an event indicating that the service may have completed. Again, it is up to the processing subsystem to decide if and how to operate based on such an event. In our case, once such an event reaches the processing subsystem, the system indicates to the interacting student that the registration requirements have been met and no further registration to courses is needed.

This interaction between the conceptual subsystem and the processing subsystem is supported by a system component we term the *Mediating Component*. The component facilitates routing of notification events to the processing subsystem. In the figure, it can be seen that the component has procedures dedicated to handle notification events from the conceptual subsystem (such as, *HandelConstraint-Event*). These procedures activate the relevant methods at the processing subsystem as an outcome of the events.

Thus, formally we propose a system architecture comprising four components (see Fig. 3):

– *The Conceptual Subsystem (CS)* A subsystem implementing REDK knowledge. The source code of this subsystem includes explicit representation of REDK by utilizing the REDK representation constructs base classes. The knowledge incorporated in this subsystem can be used by processing subsystems, defined in the next paragraph.
– *The Processing Subsystems (PS)* A set of subsystems incorporating all other system-related issues such as data type and structure information, service processing, algorithmic details, input/output operations, device interaction, and user interface aspects. Within the source code of these subsystems, no segments would embed REDK.
– *The Instance Base (IB)* An instantiation of domain-related elements, documented and coded at the Conceptual Subsystem. In other words, the IB incorporates instances of specific actors, roles and resources, of which knowledge needs to be used during runtime of the specific system. The source code defining these elements is found at the CS, and upon system execution, the PS can instantiate these elements. Once instantiated, these instances are found at the IB.
– *The Mediator Component (MC)* A component which incorporates all knowledge associated with interaction between other components. This component does not include domain knowledge. It only facilitates the interaction between the PS and its IB.

6 Discussion

Our method addresses consolidating agile processes with domain documentation. We have developed an *active* documentation approach, aimed at capturing the conceptual model of the domain. Conceptual models have been well recognized as facilitators of communication and domain understanding. Also, while system models are represented to different extents in actual implementation code, domain models are practically lost once implementation takes place.

In order to enable explicit documentation of domain knowledge typically found in domain models, use of core constructs from the different modeling approaches needs to be supported. In this work, we have identified the core constructs for domain models, but our approach is not limited only to these constructs. As research progresses and new representation constructs are identified, their use can be integrated as well.

Indeed, extended representation constructs have been suggested in methodologies that evolved. The set of extended constructs that are relevant for representation may vary in different organizational settings. Other representation constructs can be incorporated by defining their meaning and their relations with the identified core constructs, and accordingly defining respective core classes for use in IS code.

Our documentation approach is adaptive and fits within the context of agile development. According to Cockburn [17], the agile manifesto incorporates four core values motivating agile software development practices. All values are kept when using the ADSD approach:

- *Individuals and interactions over processes and tools* Agile development emphasizes professionals who comprise a project team, rather than specific tools. The ADSD complies with this principle, as professionals are needed to model the domain. ADSD is based on the modeling by people and does not impose specific tools or processes. ADSD assures that the system complies with the environment understanding.
- *Working software over comprehensive documentation* As previously mentioned, agile methods do not emphasize documentation. One of the core motivators for the ADSD architecture is to enable documentation that is in fact part of the executable system. Therefore, ADSD proposes a way of consolidating this principle with the need for documentation.
- *Customer collaboration over contract negotiation* Agile teams follow practices that keep them focused on the needs of their customers. Agile development advocates only contractual relationships that encourage the project team to work with its customer. In order to support such values, proper communication with customers should be established. ADSD supports such communication by enabling developers to understand the customer domain.

- *Responding to change over following a plan*
  Agile development stresses practices that enable adjustment to changing requirements and system environments. An important element of ADSD is its supporting architecture, under which domain knowledge is isolated from implementation knowledge. In ADSD, changes in knowledge about the IS environment propagate into the conceptual subsystem. This propagation has two effects: It facilitates communication about the changing environment, and it immediately affects the system operation.

7 Summary

This study concerns documentation in agile development processes. Agile software development methods attempt to offer an answer to the eager business community, asking for lightweight and nimbler software development processes [3]. Agerfalk and Fitzgerald [4] claim that these development methods differ significantly from the traditional plan-based approaches by emphasizing development productivity rather than the rigor process.

In a parallel venue, there is a general agreement that the major disadvantage of agile processes is in the loss of undocumented knowledge [3]. Moreover, this knowledge is important to support communication, which is crucial in agile development [2, 17, 39]. Furthermore, due to the general trend toward globalization, documentation becomes increasingly important since in physically separated development teams, communication becomes even more difficult, especially in agile development processes [30, 31]. In this context, Parnas [52] suggests that agile methods that try to avoid documentation should not be recommended.

Since agile processes take the general stand that true design can only be found in working source code, we have analyzed the relations between implementation and design documents. We have found that traditionally documented knowledge, which is not readily available in system source code, is the knowledge accumulated during the process of requirements engineering. Both agile RE and traditional RE processes come to help elicit user requirements. However, while traditional RE stresses rigid processes and documented domain knowledge, the agile processes forfeit domain documentation, assuming such documentation cannot effectively evolve.

By analyzing traditional RE processes, we are able to extract the constructs facilitating documentation of domain knowledge. We suggest that agile documentation of domain knowledge is enabled by establishing means for using these documentation constructs as part of the system code. We also suggest an architectural design in which domain knowledge is represented explicitly and is isolated from other segments of code. Under this design, the documented domain knowledge drives the processing of the system at runtime.

This study identifies documentation constructs that are missed in agile processes and provides methods for incorporating their agile use. The work presented in this manuscript paves the way for a holistic approach that can be proved by a series of experiments and practical experience. As this research suggests that agile methods can incorporate agile documentation, many directions for future research can originate from it. For example, extending our framework to support explicit documentation of other types of knowledge in agile processes is one direction for future work. Alternatively, examining other possible mechanisms and architectures to support the use of documentation constructs within working system code is another possible venue for future research.

Acknowledgments The authors thank Yair Wand and Andrew Burton-Jones of UBC for their helpful comments.

References

1. Abbot RJ (1987) Knowledge abstraction. Commun ACM 30(8):664–671
2. Abrahamsson P, Salo O, Rankainen J, Warsta J (2002) Agile software development methods—review and analysis. VTT Electronics
3. Abrahamsson P, Warsta J, Siponen MT, Ronkainen J (2003) New directions on agile methods: a comparative analysis. In: Proceedings of the 25th international conference on software engineering 2003, Portland, Oregon, pp 244–254
4. Agerfalk PJ, Fitzgerald B (2006) Flexible and distributed software processes: old petunias in new bowls? Commun ACM 49(10):27–34
5. Alford MW, Lawson, JT (1979) Software requirements engineering methodology RADC-TR-79-168, U.S. Air Force Rome Air Development Center, Griffiss AFB, NY (DDC-AD-A073132)
6. Anton A (1996) Goal-based requirements analysis. In: Proceedings of the second IEEE international conference on requirements engineering (ICRE'96), Colorado Springs, USA, pp 136–144
7. Balasubramaniam R, Cao L, Mohan K, Xu P (2006) How can distributed software development be agile? Commun ACM 49(10):41–46
8. Barker J, Palmer G (2004) Beginning C# objects: from concepts to code. Apress, Berkeley, CA
9. Bennett K, Cornelius B, Munro M, Robson D (1991) Software maintenance. In: McDermid JA (ed) Software engineer's reference book. Butterworlh-Heinemann, Oxford
10. Booch G (1987) Software engineering with ADA. Benjamin-Cummings, Redwood City
11. Castro J, Kolp M, Mylopoulos J (2002) Towards requirements-driven information systems engineering: the Tropos project. Inf Syst 27(6):365–379
12. Chen PP (1976) The entity-relationship model—toward a unified view of data. ACM Trans Database Syst 1(1):9–36

13. Chen X, Jin Z, Yi L (2007) An ontology of problem frames for guiding problem frame specification. In: Proceedings of the 2nd international conference on knowledge science, engineering and management (KSEM 2007), Melbourne, Australia, pp 384–395
14. Chung L, Nixon B, Yu E, Mylopoulos J (2000) Non-functional requirements in software engineering. Kluwer, Dordrecht
15. Clarke S, Harrison W, Ossher H, Tarr P (1999) Subject-oriented design: towards improved alignment of requirements, design and code. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '99), Denver, Colorado, United States, pp 325–337
16. Coad P, Yourdon E (1991) Object-oriented analysis. Yourdon, Upper Saddle River
17. Cockburn A (2002) Agile software development. Addison Wesley, Boston
18. Dardenne A, Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Program 20:3–50
19. Davis AM (1988) A taxonomy for the early stages of the software development life cycle. J Softw Syst 8(4):297–311
20. Devedzik V (2002) Understanding ontological engineering. Commun ACM 45(4):136–144
21. Dubois E, Hagelstein J, Lahou E, Rifaut A, Williams F (1986) A knowledge representation language for requirements engineering. Proc IEEE 74(10):1431–1444 (special issue on knowledge representation)
22. Fowler M (2003) The new methodology. http://www.martinfowler.com/articles/newMethodology.html
23. Fox MS, Barbeceanu M, Gruninger M (1995) An organization ontology for enterprise modelling: preliminary concepts for linking structure and behaviour. Comput Ind 29:123–134
24. Grady JO (2006) Systems requirements analysis. Academic Press, London
25. Halpin T (1998) Object role modeling (ORM/NIAM). In: Bernus P, Mertins K, Schmidt G (eds) Handbook of architectures of information systems. Springer, Berlin, pp 81–101
26. Hall JG, Rapanotti L, Jackson M (2005) Problem frame semantics for software development. J Softw Syst Model 4(2):189–198
27. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8:231–274
28. Hay DC (2003) Requirements analysis: from business views to architecture. Prentice Hall, Upper Saddle River
29. Henderson-Sellers B, Edwards JM (1990) Object oriented systems life cycle. Commun ACM 33(9):142–159
30. Herbsleb JD, Atkins DL, Boyer DG, Handel M, Finholt TA (2002) Introducing instant messaging and chat into the workplace. In: Proceedings of the ACM conference on computer-human interaction, Minneapolis, pp 171–178
31. Herbsleb JD, Mockus A (2003) An empirical study of speed and communication in globally distributed software development. IEEE Trans Softw Eng 29(6):481–494
32. Hermann K (1999) Difficulties in the transition from OO analysis to design. IEEE Softw 16(5):94–102
33. Highsmith J, Cockburn A (2001) Agile software development: the business of innovation. Computer 34(9):120–122
34. Jackson MA (2001) Problem frames: analyzing and structuring software development problems. Addison-Wesley, Boston
35. Jacobson I (1995) A confused world of OOA and OOD. J Object Orient Program 8(5):15–20
36. Kavakli E (2002) Goal oriented requirements engineering: a unifying framework. Requir Eng J 6(4):237–251
37. Kavakli E, Loucopoulos P (2003) goal driven requirements engineering: evaluation of current methods. In: Proceedings of the 8th CAiSE/IFIP8.1 workshop on evaluation of modeling methods in systems analysis and design (EMMSAD 2003), Velden
38. Kovitz B (2003) Hidden skills that support phased and agile requirements engineering. Requir Eng 8(2):135–141
39. Korkala M, Abrhamsson P, Kyllonen P (2006) A case study on the impact of customer communication on defects in agile software development. In: Proceedings of AGILE 2006, Washington, DC, pp 76–88
40. Laitinen K (1992) Document classification for software quality systems. ACM SIGSOFT Softw Eng Notes 17(4):32–39
41. Laitinen K (1996) Estimating understandability of software documents. ACM SIGSOFT Softw Eng Notes 21(4):81–92
42. Lamsweerde A (2000) Requirements engineering in the year 2000: a research perspective. In: Proceedings of the 2000 international conference on software engineering, Limerick, Ireland, pp 5–19
43. Lamsweerde A (2001) Goal-oriented requirements engineering: a guided tour. In: Proceedings of the 5th IEEE international symposium on requirements engineering, Toronto, pp 249–262
44. Liu L, Yu E (2001) From requirements to architectural design—using goals and scenarios. In: From software requirements to architectures workshop (STRAW 2001), Toronto, pp 22–30
45. Loucopoulos P, Kavakli V (1997) Enterprise knowledge management and conceptual modelling. In: Workshop on conceptual modeling: current issues and future directions. Springer, Los Angeles, pp 45–79
46. Mylopoulos J (1992) Conceptual modeling and Telos. In: Locoupoulos P, Zicari R (eds) Conceptual modeling databases and CASE. Wiley, New York, pp 49–68
47. Mylopoulos J, Borgida A, Yu E (1997) Representing software engineering knowledge. Autom Softw Eng 4(3):291–317
48. Nakajo T, Kume H (1991) A case history analysis of software error cause-effect relationships. Trans Softw Eng 17(8):830–838
49. Nawrocki J, Jasiński M, Walter B, Wojciechowski A (2002) Extreme programming modified: embrace requirements engineering practices. In: 10th anniversary joint IEEE international requirements engineering conference (RE'02), 303 pp
50. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering (ICSE 2000) Limerick, Ireland, pp 35–46
51. Paetch F, Eberlin A, Maurer F (2003) Requirements engineering and agile software development. In: Proceedings of the twelfth IEEE international workshops on enabling technologies infrastructure for collaborative enterprises (WETICE'03), Linz, pp 308–313
52. Parnas D (2006) Agile methods and GSD: the wrong solution to an old but real problem. In: Agerfalk PJ, Fitzgerald B (eds) Flexible and distributed software processes: old petunias in new bowls? Commun ACM 49(10): 27–34
53. Petri CA (1962) Kommunikation mit automaten. PhD dissertation, University of Bonn
54. Petterson J (1977) Petri nets. ACM Comput Surv 9(3):223–252
55. Reeves WJ (1992) What is software design? C++ J 2(2)
56. Rolland C, Prakash N (2000) From conceptual modeling to requirements engineering. Ann Softw Eng 10:151–176
57. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W (1991) Object-oriented modeling and design. Prentice-Hall, Upper Saddle River
58. Shlaer S, Mellor SJ (1992) Object lifecycles: modeling the world in states. Yourdon Press, Upper Saddle River
59. Sommerville I (1989) Software engineering. Addison-Wesley, Wokingham
60. The Agile Manifesto (2001) Manifesto for agile software development. http://www.agilemanifesto.org
61. Turner JA (1987) Understanding the elements of system design. In: Boland RJ, Hirschheim RA (eds) Critical issues in information systems research. Wiley, New York, pp 97–111
62. Uschold M, King M, Moralee S, Zorgios Y (1998) The enterprise ontology. Knowl Eng Rev 13(1):31–89

63. Welsh J, Han J (1994) Software documents: concepts and tools. Softw Concepts Tools 15(1):12–25
64. Yu E (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceeding of the third IEEE international symposium on requirements engineering, Annapolis, pp 226
65. Yu E (2001) Agent-oriented modelling: software versus the world. In: Agent-oriented software engineering AOSE-2001 workshop proceedings, pp 206–225
66. Yu E, Mylopoulos J (1994) Understanding "why" in software process modelling, analysis, and design. In: Proceedings of the 16th international conference on software engineering, Sorrento, Italy, pp 159–168
67. Zachman JA (1987) A framework for information systems architecture. IBM Syst J 26(3):276–292
68. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM Trans Softw Eng Methodol 6(1):1–30