Adaptive development and maintenance of user-centric software systems

Claus Pahl
School of Computing
Dublin City University
Dublin 9
Ireland
phone: ++353 +1 700 5620
fax: ++353 +1 700 5442
email: Claus.Pahl@dcu.ie

Abstract
A software system cannot be developed without considering the various facets of its environment. Stakeholders – including the users that play a central role – have their needs, expectations, and perceptions of a system. Organisational and technical aspects of the environment are constantly changing. The ability to adapt a software system and its requirements to its environment throughout its full lifecycle is of paramount importance in a constantly changing environment. The continuous involvement of users is as important as the constant evaluation of the system and the observation of evolving environments. We present a methodology for adaptive software systems development and maintenance. We draw upon a diverse range of accepted methods including participatory design, software architecture, and evolutionary design. Our focus is on user-centred software systems.

Keywords: Adaptive development and maintenance, Requirements and software change, Participative design, Software architecture, Evolutionary design.

1. Introduction

A software system is always tightly embedded into its environment. Its organisational and technological environment determines substantial parts of the system. Stakeholders influence a system throughout its lifetime. Change is ubiquitous in all types of environments. For a software system this means to adapt to the needs of different stakeholders and to adapt to constant change in its environment. We propose here a methodology for adaptive development and maintenance of user-centric software systems. We focus our investigation on interactive software systems, as these software systems are particularly dependent on their links with their environments. Interactive systems enable users to communicate with the system [7].

Our focus is on requirements engineering aspects for software systems that are strongly influenced by two characteristics: user-centric and change-driven, both representing a software system's links to its environment. User-centric systems are usually interactive systems. We aim at a participative form of software development that also includes the user in all stages of the software lifecycle such as maintenance and change management in evolving environments. Change can be a consequence of internal evaluation or of evolution in the environment.

Classical approaches to requirements engineering, such as use cases, have shown deficiencies [13]. Use cases, for example, are difficult to formalise and to manage on a large scale; change management is often a problem. Solutions to these specific problems include distinguishing soft and rigid requirements, to use goals to structure use cases into a hierarchy, or to use an incremental approach. These approaches lead us already towards an iterative style of development based on possibly change- and evolution-oriented increments and changes. Adaptiveness is our central notion, the paramount ability within the development and maintenance process that captures the reaction to the various forms of changes resulting from evaluations and evolution in relation to a system's environment. Our objective is to develop a methodology that focuses on a software system's links to its environment and that allows the system to be adapted to changes in its environment.

Our contribution to adaptive development and maintenance is a combination of widely used software engineering methods that we have adapted to the given requirements engineering context. We have combined participative, architectural and evolutionary design [4,16,21] with a focus on aspects of volatile, emerging and changing requirements – supported by formative evaluation and evolution in incremental and cyclical processes. Scenarios form the starting point, reflecting activity-based requirements for the system and its development process [3,4,9,25]. Scenario prototyping, i.e. operationalising scenarios, emphasises the evaluation and evolutionary focus [2]. We will show the feasibility of such a combination. Adaptive development and maintenance is based on architecture-based evolutionary scenario prototyping. The central achievement is the combination of usability-oriented techniques, such as scenarios, prototyping, and evaluation, with architecture and software change techniques.

We start by giving an overview of adaptive development and maintenance, before introducing the three central aspects in detail. We illustrate the methodology using a case study – an educational system that exhibits all the difficulties of the target domain. Finally, we evaluate our methodology and end with some conclusions.

2. Adaptive development and maintenance – overview and rationale

2.1 A notion of adaptiveness for software systems

To adapt means to change your (or a system's) behaviour because your (or the system's) situation or environment has changed. Software systems are changed or adapted so that they can be used in changing environments, adapted to the needs of different stakeholders. The environment here comprises the technical and organisational environment – the computer system – in which the software is running, but also the stakeholders involved and their needs.

Adaptiveness is different from evolution or development in general. Evolution is about change, but based on the idea of improvement starting from something elementary. Development, equally, has the connotation of growth and increase. Adaptiveness is the broader term, which captures all forms of maintenance and change, and is not restricted to ideas of improvement and growth; it represents *flexibility in accommodating change within a system*. *Adaptiveness* is a property that refers to the flexible adaptation of software systems in changing environments. It has a *static dimension* relating to stakeholders and the system environment and a *dynamic dimension* relating to evolution and change processes.

In the context of software systems adaptiveness involves two aspects that bring us back to the central aspects user (and other stakeholders) and change. We introduce here a methodology – called *adaptive development and maintenance* – that encompasses *requirements engineering* and *design* aspects, based on *participative* and *evolutionary design*, both connected through *software architecture* [4,16,21]. We present this methodology as a framework based on these individual methods, connected through common rules and principles.

Change and *evolution* make up the first focus of our methodology. A goal is to incorporate a design that allows a software system to be managed and maintained in a changing and evolving environment. Evolution results in changes in two directions: improvements of existing features and extensions of the existing feature range. Evolution and change raise questions about feasibility, effectiveness, and potential conflicts. Prototyping and analyses provide the answers [24]. Experimental prototyping is the proposed method to investigate the feasibility of solutions for emerging or changing requirements. Questions about efficiency and other quality of service criteria can be answered by analyses and exploratory prototyping. An analysis of existing and modified requirements and design decisions is needed to determine possible conflicts. On a smaller scale, change increments and change iterations are key concepts. These are at the core of the implementation of evolution, but they are also central in general maintenance and change management. Internally driven maintenance and change is based on evaluations aiming at improved software productivity. A goal of the methodology is the traceability of requirements and their change. This should address local and global changes, and internal and external change factors.

The *user* – or stakeholders in general – is the second focus besides change. To adapt to the stakeholders' changing needs is the goal. In interactive systems, the user plays the key role and usability requirements are paramount. However, these requirements do often cause difficulties – we will illustrate this later on in our case study. These requirements are often volatile, i.e. subject to change or only emerging during the software product lifecycle.

2.2 An overview of adaptive development and maintenance

We propose a *three-stage iterative process model* as the core of our adaptive development and maintenance methodology consisting of the three stages *participative requirements engineering*, *adaptable architecture design*, and *evaluation- and evolution-driven change and maintenance*. The methodology, see Fig. 1, involves aspects of evolutionary and incremental development. We distinguish two layers: the *artefact layer* with scenarios, architecture, and prototypes and an *analytic layer* with evaluation and change management techniques. Analysis brings us from the artefact layer to the analytic layer; change brings us from the analytic to the artefact layer.

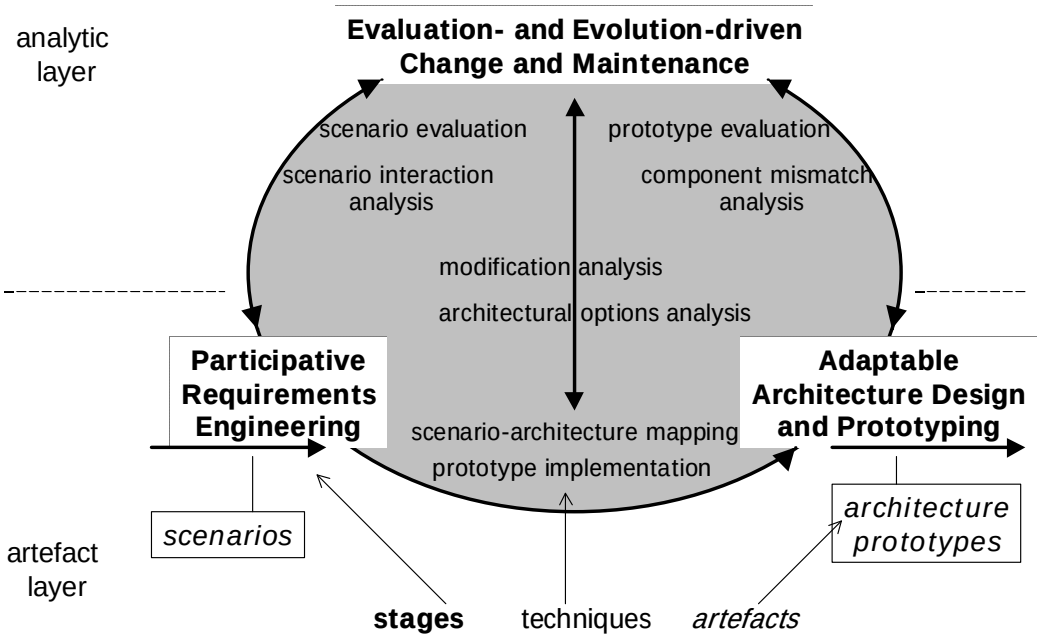

Figure 1. Adaptive development and maintenance.

A methodology is determined by the *stages*, the *artefacts* that are involved, and the *techniques* that are based on *activities* on these artefacts. We distinguish a requirements stage, a design stage, and a change and maintenance stage. Central *artefacts* for our three-stage adaptive development and maintenance are scenarios for the requirements stage, architectures and prototypes for the design stage, and prototypes also as evaluation artefacts for the change and maintenance stage. Prototypes are reflections of scenarios within the architectural structure of the software system. The *techniques* are based on *activities*: firstly, *encodings* – essentially mappings from one artefact representation notation into another – and, secondly, *analyses and evaluations* that support these mappings.

Encodings are mappings from representations in one notation or language into another target notation creating different types of *artefacts* (Fig. 1):

- The description of *scenarios* is the first step. Based on general goals and objectives of the stakeholders, scenarios can be formulated that reflect the activities of stakeholders as users of the system or within the development and maintenance process. A scenario language is the target notation.
- The definition of an *architecture* is the second step. Scenarios are the starting point for the mapping – although other factors can determine general platform and specific architectural style choices. We use an architectural description language with notational elements such as components, connectors, and interaction processes as the target language. A basic scenario is a triple (actor, action, object); action and object determine (together with an architecture style) the components. Scenario activities determine component interaction processes. There will usually be

more components and connections than determined by the scenarios alone; the latter only reflect stakeholder activities. The mapping process is guided by specific analyses, for example to assess different architectural options.

- The implementation of *prototypes* is the next step. A scenario and an architecture together determine a prototype. The prototyping language is often a programming language. Based on the architecture, the components and connectors that are affected by a scenario mapping are determined. A prototype is based on a subset of services offered by the components in question.

Artefacts and mappings are subject to change in an incremental and evolutionary process. Adapting a system to continuously occurring changes in the environment and in its requirements is supported through *analyses* and *evaluations* (Fig. 1):

- Scenarios:
  - *Scenario evaluation* with stakeholder participation reflects requirements elicitation; it is about the validation and correctness of activity requirements.
  - *Scenario analyses* address interaction and consistency – applied to any new or changing scenario. Interaction is a measure of coupling and cohesion; similarity analyses aim at the detection of overlaps through the identification of equal substructures.
- Scenario-architecture mappings:
  - An *architectural options analysis* is used in both an initial architecture design and in architectural transformations. Guidance rules and heuristics are used to select architectural options.
  - A *modification analysis* addresses changing architectures. Determining the change impact comprises the options, which components are affected by change, and how much work is needed per component. We use, among others, coupling and cohesion analyses as maintainability measures.
- Architectures and prototypes:
  - *Prototype and architecture evaluation* with stakeholder participation focuses on the completeness, correctness, and consistency of functional requirements and non-functional requirements (such as usability, efficiency, and stability).
  - Architecture assembly – initially or during change management – can be supported by a *component mismatch analysis* looking at syntactical and semantical aspects. *Change consistency* can be controlled through the use of component subtypes and flexible connector types.

2.3 Central principles: interaction processes and adaptivity

The three stages of our methodology are part of almost any development methodology. Central for the success is the internal coherence of these stages. The rationale behind the methodology should be reflected by the coherence of the stages in terms of their basic principles. Two *common principles* are central for the coherence the methodology. Both act as integrating elements.

- From a technical, product-oriented perspective, *interaction processes* embody the central focus of software systems within their environment. Interaction processes are the integrating principle for the different encoding notations and analysis techniques.
- From a development and management process-oriented view, *adaptivity* is the integrating principle for the three different stages that captures stakeholders and changing requirements as part of the organisational environment.

Managing the relationship of a software system with its environment – in the two dimensions interaction and adaptivity – is at the core of our methodology. The *rationale* is to address problems arising in the development and maintenance of user-centric software systems that can often be attributed to the system-environment relationship. *Interaction* is the central notion since it emphasises the relation of a system to its environment. This includes the technical context, but also stakeholders and their requirements on a more abstract level. Adaptivity refers to the flexible management of interaction.

User behaviour and interaction processes between user and system are central in interactive, user-centred systems. Interaction processes capture user-system interactions as well as system-internal interactions. This aspect plays a central role and integrates the three stages of the methodology. The scenario language focuses on complex behaviour processes based on simple interaction activities. Our architecture description language includes interaction process descriptions. Prototypes are often

horizontal prototypes focusing on the interaction between components. The evaluation focuses on usage mining and analysis of behaviour patterns.

We have outlined the methodology in this section. Evaluation and evolution create an iterative development process. The overall approach can be characterised as *cooperative architecture-based evolutionary scenario prototyping*. We will describe the methodology in more detail by looking at, firstly, the use of basic descriptional elements: units of description (e.g. scenarios) and combinators of descriptions (sequence, iteration, etc.), and, secondly, the use of basic process activities including the use of techniques such as mappings, evaluations, and analyses.

3. Participative requirements elicitation and representation

3.1 Requirements engineering

Requirements engineering is a *communicative process* involving stakeholders with different backgrounds, expectations, and roles in a software development project [13,24]. Requirements engineering involves the *elicitation* and the *representation* of requirements. The fundamental difference between these two activities is their focus on two different stakeholders. Requirements elicitation is generally user-centred. Informality and concreteness of requirements are important. Requirements representation is on the other hand developer-oriented. Here, consistency and completeness of requirements are central. Still, the *organisation* and *representation* of requirements is a reflection of the approach to requirements engineering, e.g. a reflection of the extent to which users are included in the development process. The *user* and her/his *environment* are therefore the central focus in requirements engineering. The context of a future domain-specific software system involves the stakeholders, the relevant properties of the domain such as terminology, standards, and domain models.

Different types of requirements can be distinguished in our context. Participation of the user and other stakeholder is important in order to make *vague requirements* more precise in an iterative communication process. *Emerging* and *volatile requirements* can be dealt with in an incremental and evolutionary approach, i.e. in a software lifecycle process that reacts to changes in the environment at later design and deployment stages. *Conflicting requirements* can emerge during development, or as a consequence of change and evolution. Requirements can be stable or volatile; they can be subject to change (they are change objects) or they can address the process of change (they are change subjects) and maintenance activities. Determining change factors is often a difficult task: what is likely to change and how can change be made explicit are the critical questions.

3.2 Scenarios in user-centred requirements engineering

3.2.1 Scenario-based design

Participative design is a user-centred development approach, in particular suitable for software systems with a high degree of user interaction and complex processes involving the user [4,9]. The focus is on usability requirements [7]. Central concepts of the approach are scenarios. *Scenario-based design* is a representational form for user-centric development [3]. Users and designers participate actively in the development process. *Scenarios* are brief descriptions of a single interaction of a stakeholder with a system – this includes classical use cases, but also maintenance and change activities [11]. Interactive software systems are characterised by dialogues between the user and the system that represent complex processes. An adequate representation of these processes through composite scenarios is, however, necessary. *Executable prototypes* play an important role; they operationalise scenario definitions. Prototypes aim to support the evaluation of usability and utility of the software system and its components. Systems with requirements in terms of usability are often subject to incremental development. Scenarios can address this problem.

Scenarios are rooted in specific situations from the domain under scrutiny [3]. Scenarios are hypothetical (make assumptions about the future) and selective (not complete); they should be connected (related to architecture and other scenarios) and assessable (allow analysis and evaluation). Scenarios are a medium of the requirements engineering stage. However, their use extends into the design and further stages. Scenarios are part of the incremental development and maintenance lifecycle of a software system [25].

3.2.2 Scenario language

Kazman et.al. [11] define a *scenario* as a brief description of a single interaction of a stakeholder with a system. A more precise definition is given by Alspaugh et.al. [1]: a *scenario* is a linear sequence of events, with associated attributes. An event is an association of an actor and an action. A subsequence is a sequence of one or more events part of a scenario. An episode is a named subsequence, usually shared among several scenarios. Attributes of a scenario can include system goals, contractual obligations, the concreteness level, or the author. These attributes would bring the scenario description closer to a detailed design specification.

We refine the definition by Alspaugh et.al. [1]. We do not require a linear sequence. Instead, we allow a richer set of combinators to built composite activities, reflecting the interaction processes of typical user-centred software systems where a user can choose between options, can repeat elements, or work on several elements at the same time. We also expand the notion of events, calling it an activity. An activity shall here be comprised of an actor, an action, and the object on which the action is carried out. Information access is represented through the combination of action (access operations) and object (information objects). Thus, we define the following *scenario language*:

- A *basic activity* is a triple (*actor*, *action*, *object*) consisting of an *actor* (a stakeholder) who carries out an *action* on an *object*. An activity describes an interaction of an actor with an object.
- *Activity combinators* compose more basic interaction activities to more complex ones. *Option* and *repetition* can be applied to a single activity. *Choice*, *concurrency*, and *sequence* combine two or more activities.

This semi-formal language is useful in the context of interactive systems where user processes and interactions of actors with a software system have to be expressed. There has been a trend recently to add such behavioural specifications to requirements notations [12]. Composable activities allow a refinement process, starting with abstract activities refined by more detailed composite activities.

Change is a central activity in software development and maintenance. Scenarios can reflect activities of all stakeholders, including change-related and other maintenance activities in relation to the software system and its specifications. We can distinguish *usage scenarios* (or simply scenarios) and *meta scenarios* – called direct and indirect scenarios elsewhere [11]. Meta scenarios provide guidance for the maintenance process; they usually include actions such as evaluation, evolution observation, and analysis activities.

4. Adaptable architecture design and prototyping

4.1 Software architecture and architectural design

Software architecture is a software engineering discipline that addresses the organisation of software systems into composable software entities forming a software architecture of *components* and *connectors* between components [6,21]. These software components are loosely coupled, internally coherent software artefacts that are assembled to software systems. Connectors are entities that provide the infrastructure for interactions between components. Architectural design is usually the first design step before the focus narrows down onto detailed design [14]. The main focus of software architecture design is the separation of computation and communication, which enhances the maintainability of evolving systems.

4.2 Scenario-driven architectural design

Two aspects in the context of *architectural design* are of particular importance here. Firstly, the transition from requirements engineering into the architectural design stage: requirements representations can be used to determine architectures. Secondly, the effect of software ageing and evolution on software architectures: for instance, layered architectures promote independence and help to control the impact of change. However, in relation to traceability of change more support is needed. Scenarios turn out to provide a solution for both aspects. *Scenarios* can form the starting point for an architecture definition. The central principle – interaction processes – will enable the seamless transition from scenarios to architectures. Scenarios can be subject to maintenance and change; they

can consequently be a tool to trace changes, but scenarios can also address change and adaptivity explicitly.

Scenarios can be the main drivers to capture architectural views [15]. The structural view addresses architectural styles and patterns. The functional view addresses how the system realises critical functionalities (which are expressed through scenarios). The static view addresses the mapping between scenario activities and components – an important view supporting traceability, cohesion and coupling.

4.2.1 Architectural description language

Our target language for the architecture definition is an *architectural description language* [6,21]:

- *Components* encapsulate the computational side. Components provide a service based on a range of coherent operations.
- *Connectors* represent the communications infrastructure. Glue code enables the communication between component services.
- *Interaction processes* describe protocols that coordinate the interactions between the components.

Architectural description languages comprise interface definition language aspects that describe the functionality of a service in abstract syntactic and semantic form. The topology of components and connectors defines the architectural configuration – often based on an architectural style or pattern. We have added interaction processes to the language, which implements the important interaction principle.

4.2.2 Scenario-architecture mapping

Scenarios are our central requirements representation notation. They are therefore the main input for an architecture definition. We need to distinguish two aspects when mapping scenarios onto an architecture. Firstly, an initial architecture design is required when a software development project is started. Secondly, changes to an existing architecture, i.e. an architectural transformation, is required if requirements change, either due to external factors (evolution) or internal improvements (evaluation). Both activities can be guided by suitable analyses that we will address later on in Section 4.2.3.

The architecture definition is, however, also determined by other aspects. Platform decisions and standards, such as the Web, with their infrastructure impose architectural constraints. Some of the components, connections, and interactions are not determined by scenarios – such as backend aspects in information systems. Infrastructure services that are required predetermine some aspects of the architecture: scheduling and coordination of system activities, how data moves through the system, security, fault propagation, integration of new components, how the system scales, etc. [11].

Our *scenario-architecture mapping* is centred around the mapping of activity triples and complex scenario activities to components and interaction processes:

- Scenarios can be mapped onto an *initial architecture* as follows. Actors and objects of the triples support the identification of components. Scenarios can be categorised according to their interactions with others. This might indicate implementation through the same component based on similar features addressed in the scenario. Based on a scenario categorisation, the next task is to find an architectural style that supports the desired component connections and interactions. Standard architecture types are available that support particular types of software systems. Composite scenarios involving complex activities determine component interaction processes. The literature, e.g. [12], suggests mapping scenario activities onto interactivity design notations. We map the activities onto component interactions, which provides a seamless and traceable transition from requirements to design notations.
- In an evolving system, newly added or changing scenarios need to be mapped onto an *existing architecture* [15]. Components, connectors, and interactions that are affected by a scenario implementation need to be identified as part of the mapping. This mapping can be analysed to determine the change impact – this includes conflicts in requirements in relation to other scenarios or components, an estimate of the expected work needed, the components affected, and the replacement strategy.

4.2.3 Analysis

The software architecture field comprises various architecture analysis methods. Some of these can support the scenario-architecture mapping. Analyses are needed to address architectural options and to detect interferences between scenarios, prototypes, and components (Fig. 1). The early identification of conflicts and their resolution is a central objective in software development. *Mismatch analysis*, suggested by [8], and *conformance analyses* are possible techniques. A mismatch is an incompatibility between components, e.g. related to interaction patterns. The solution to the mismatch problem is the generation of wrapper or glue code. An objective is to exclude nonviable component configuration options at an early stage. This is a two-level approach:

- Scenario-oriented: At the *architectural options level*, scenarios are described and component descriptions include characteristics such as the degree of concurrency, distribution, encapsulation, reconfigurability, type of control unit, etc. Most of these are semi-formal descriptions. The *mismatch analysis* is guided by mismatch rules for scenario activities (e.g. 'sharing data might cause conflict'). Possibly conflicting components and their underlying scenarios are identified. Unsuitable options are discarded or glue code is used as a remedy.
- Architecture-oriented: At the *architecture level*, a *conformance analysis* can be carried out. The analysis of detailed, more formal specifications of the architecture is here the objective. Conformance is essentially a matching analysis between component (and prototype) interfaces. *Message communication theories* and *process algebras* can be used to describe and analyse conformance of our component interaction processes [21], which are based on standard process notations.

4.3 Architecture-based scenario prototyping

As the design process progresses, the purpose of scenario usage can change. The initial scenarios have a major impact on the architectural design. However, as the architecture stabilises, the scenario focus moves from 'typical' (i.e. architectural support patterns and reuse on the architecture level) to 'critical' as prototypes develop horizontally (to support interactions and activities across components) and vertically (to support the development of a single feature). A *prototype* is an executable scenario, determined by the mapping of the scenario onto the architecture. In addition to a scenario-based architecture definition, we propose the execution and evaluation of scenarios through experimental and explorative prototypes to address their validation. Prototypes serve as the communication medium between users and developers and scenarios act as test cases for prototypes and the architecture.

4.3.1 Prototyping language

The prototypes need to be embedded into the scenario-based architecture. A prototyping language is an implementation language that allows the rapid development of applications through specific language constructs – such as a rich and flexible type system or domain-specific, high-level libraries and APIs. The language needs to consider the structural constraints imposed by the architecture definition. The language needs to support, or interface, services of the architecture platform.

4.3.2 Scenario-prototype mapping

A prototype is identified by mapping a scenario onto the architecture. The scenario-architecture mapping identifies the components that implement a scenario. A prototype is therefore based on a subset of the services provided by these components.

Implementing usage scenarios through prototypes supports traceability of requirements and their changes. Prototypes aim to operationalise scenarios, in particular the activities described in scenarios, but they also incorporate *design and implementation decisions*. Standard *mechanisms*, based on engineering design principles addressing performance, fault-tolerance, etc., are used to implement the scenario. These mechanisms determine the internal structure and behaviour of the prototype within the constraints set by the scenario definition. The mechanisms need to be inherently linked to architectural mechanisms and properties, such as architectural styles [11].

5. Evaluation- and evolution-driven change and maintenance

5.1 Factors of software change

High frequency and variety of change characterise a wide range of software systems. These change factors might be

- *internal* as a result of *evaluations* that help to eliminate faults and to improve – these evaluations can address scenarios, prototypes, and architectures,
- *external* as a result of *changes and evolution* in volatile environments, which can affect scenarios, prototypes, and architectures.

Causes and forms of change are manifold; their implementation through software change techniques requires the ability to adapt to changes supported by suitable methods and supporting analyses.

5.1.1 Scenario and prototype evaluation

Scenarios are constructions meant to stage activities in the future and to reflect on and illustrate problems with these activities. They serve to predict and evaluate the user's actions in the system. The evaluation of scenarios in collaboration between developers and users is one of our key objectives. The purpose of communication and collaboration is to elicit and validate requirements and to establish their correctness.

Recently, in the literature, a shift could have been observed from abstract scenario evaluation to *executable scenarios* (prototypes). Prototypes create a trial-use situation to allow users hands-on experience with the future system. Consequently, the prototypes create user reflections for feedback to the developer. Scenario-based prototypes become a means for a *structured formative evaluation* of system properties – the abstract scenario descriptions are test cases for the prototypes.

Prototypes bring requirements to the architectural design and implementation level reflecting scenarios in the architecture. The interaction aspect provides the central integration between the different encodings. Prototypes help us to evaluate aspects beyond the textual or graphical descriptions of scenarios and architectures by directly analysing specific aspects of a running system. They allow us to address qualitative and quantitative system properties such as usability, reliability or the degree of feedback – usage and usability are central here. Prototypes allow the identification of deficiencies, for example through breakdown analysis [2]. Prototypes in a changing environment help us to capture requirements emerging during the development and deployment; they allow an incremental approach to software development that can deal with changes; and they support the assessment of necessary changes to the architecture and implementation. Classical techniques, such as surveys, observation, and quality of service measures, combined with advanced usage analysis techniques can be used to *evaluate a prototype*.

5.1.2 Scenario and architecture evolution

Software evolution occurs as a consequence of changes in the environment, i.e. changes to functional or organisational requirements and/or infrastructure technology. Dealing with evolving scenarios and architectures requires suitable management and maintenance techniques.

Evolutionary design is an answer to these management and maintenance problems. The driver of evolutionary design is change. Central problems are vague, volatile and emerging requirements. The problems arising from this context are addressed through adaptive software development concepts such as architectures, analytic models, and scenarios [5,16,17]. In an environment dominated by change, *analysis* becomes an important adaptivity technique for the management of software evolution and evaluation. Requirements traceability and consistency need to be analysed and evaluated. Prototyping is an approach that helps in this process of validating requirements, e.g. to establish the completeness, correctness, and consistency of requirements.

5.2 Analysis for software change

Whether changes are caused internally (evaluation) or externally (evolution), the development and maintenance method has to accommodate handling of these changes [20]. *Traceability* and *consistency* are two key requirements for change support. Changes need to be traced from requirements through design to implementation descriptions – we made this feasible for our methodology based on the coherency achieved through the central interaction principle. Consistency is maintained through

conflict identification and resolution. In general, the central issues that determine a suitable *software change* technique are

- the *sources* that initiate change (e.g. programmed or ad-hoc),
- the *operations* that implement change (e.g. add or remove a component or connector),
- the *constraints* that control change (e.g. to preserve structural or behavioural properties), and
- a *language* to express modifications (procedural) and constraints (declarative).

The evolution of software architectures (design stage) needs to be integrated with scenario evolution (requirements stage). At the core, the *change impact* needs to be determined. Scenarios and prototype evaluation provide a suitable starting point. Both evaluation and evolution might entail software changes. A *change impact analysis* needs to determine what the options for change are, how each change option can be realised, and what the costs associated with each change option are.

5.2.1 Analytic models

Scenarios and architecture definitions are concrete artefacts produced in the development process. Scenarios are abstractions of stakeholder's tasks and activities, involving the information objects that are accessed. Often, these artefacts are based on underlying, explicit or implicit models that support design and evaluation.

- A *domain model* captures concepts from the application domain and their properties.
- A *conceptual model* is system oriented, consisting of an information model (data structures) and a behaviour model (interaction behaviour).
- An *analytic model*, which combines domain and conceptual models, serves to interpret evaluation results. Scenario activities as the basic descriptional units are based on these models.

Using scenarios and prototypes, we build analytic models of a use of, or of a change to, the architecture. This allows us to understand the impact of the activity described in the scenario on the architectures. This analytic model allows us to evaluate the required quality attributes (performance, security, modifiability, etc). It also allows us to address the impact of change in an existing system.

5.2.2 Scenarios

We can use scenarios – both new or updated usage scenarios and meta scenarios – to guide the analysis of the impact of change on the architecture. This helps us to identify components that might need to be modified in case of changes in requirements. In combination with software architecture, scenarios and prototypes support traceability at the architectural level. In more detail, *scenario analysis* can comprise:

- *Scenario interaction analysis.* This technique, based on the detection of overlapping sets of objects or actions defined in the scenario activity triples, gives ideas about coupling and cohesion, which in turn helps to assess the impact of change (similar to mismatch analysis at the architectural level).
- *Scenario relationships and similarity*. Episodes (interaction processes) are building blocks of scenarios. Sharing the same episodes can quantify the similarity of scenarios. The detection and analysis of change in similarities for a software system yields an evolution impact measure [1].

Scenario analysis is a central technique to assess the modifiability of system designs [5].

5.2.3 Scenario-architecture mapping

The scenario-architecture mapping is an important specification documenting design decisions. Analyses based on the scenario-architecture mapping can comprise:

- *Modification analysis*. Usually, several options for change exist. It has to be analysed and determined what to pursue and what to discard – the impact of change on the architecture is the main criterion.
- *Coupling and cohesion measures*. The measures can provide input for modification analyses – these were already discussed earlier on.

Similar to scenarios and the architecture, the mapping is not static, but subject to (controlled) change.

5.2.4 Architecture

Software architecture is also a maintenance tool. Component-based architectures provide a separation of computation and communication, i.e. a software system is described in terms of its components (encapsulating computation behind interfaces) and its connectors (the connections between components that allow them to communicate). This technique helps us to support reconfigurability and reuse of

software. Composition is interaction; architectures composed of independent, interacting components create independence, which supports maintainability and modifiability in changing environments.

The implementation of change affects the software architecture. A *subtype notion* can guide controlled evolution and architectural transformation [16]. Types comprise interface syntax and semantics such as interaction behaviour. Evolution is the preservation of architectural type correctness. For example, the preservation of behaviour such as interaction processes can be guaranteed. Interaction processes are patterns (including for instance optional interaction activities) that allow changes up to a certain degree.

6. Case study

Our focus on user-centric systems reflects a type of software systems that we have been involved in for a long period of time. Our case study software system is an *interactive educational multimedia system*. We have also been involved in the development and management of other user-centric systems such as e-commerce systems. We have chosen the education domain over other application areas, since it is characterised by complex and highly interactive usage processes. The case study is additionally characterised by a long evolution and maintenance history.

From a design perspective, the case study system is characterised by the following aspects: an *information model* that comprises data representation and storage, in particular knowledge (declarative and procedural) for the educational application domain is important, an *interface* to support user activities that are part of learning dialogues and processes, and a component-based *architecture* implemented on a Web platform and consistent with a domain-specific standard. The main stakeholder in the system are teachers and students as the end-users of the system, the instructional designers (for the educational elements), and the software developers and administrators (essentially comprising development, deployment and maintenance support) as the developers and managers of the system.

The system – a virtual database course – exists since 1996 based on the current Web platform [23]; an older version on a pre-Web hypertext platform had been in use since 1991 [22]. The system has been developed in a sequence of major phases in which substantial targeted features were realised [19]. Substantial investments for major improvements (evolution steps) were necessary for these phases; maintenance and incremental changes were financed within the given budget. These major evolutions were accompanied by various changes resulting from the day-to-day business of running the system and from system evaluations that were carried out internally or in collaboration with the stakeholders. Adapting the system to new circumstances has often been a major challenge.

6.1 Participatory requirements elicitation and scenario representation

Table 1 lists a few scenarios to demonstrate their variety in describing stakeholder activities. Scenarios are described by activitity triples – comprising actor, action (possibly composite), and object – and the scenario type.

Actor	*Action*	*Composite*	*Object*	*Type*
Student	Login	No	ID + password	Usage
Student	Download	No	Learning resources	Usage
Student	Streamed lecture	Yes	Audio + HTML	Usage
Student	Animated tutorial	Yes	Flash animations	Usage
Student	SQL query exercise	Yes	Interactive database access	Usage
Student	Self-assessment	Yes	Multiple choice questions	Usage
Teacher	Evaluate	Yes	User activity log	Usage
Instructional designer	Upload content unit	No	Learning resources	Meta
Software developer	Link maintenance	No	HTML	Meta
Software developer	Update delivery system	Yes	Delivery system	Meta
Software developer	Add new feature	Yes	Delivery system + learning resources	Meta

Table 1. Scenarios for the teaching and learning environment.

The scenarios cover user (student, teacher) and developer (instructional designer, software developer) activities. Scenarios reflect the requirements elicitation process. Students were asked about the activities they would like to be supported and about the quality of service they would expect. Teachers were concerned with the evaluation system providing feedback about student activities in the system, assuming that content and runtime support was provided by other stakeholders (instructional designer and developer). Facilities to upload and integrate material were required by instructional designers. Software developers required a clear specification of the activities to be implemented and their tasks in relation to the maintenance of the existing system.

An abstract usage scenario – called SQL query exercise – is presented in Fig. 2. It describes a composite activity consisting of four basic activities combined to a composite activity using sequence and repetition. This composite activity can be seen as a refinement of the abstract activity from Table 1. Each of the basic activities represents an information access activity – an actor interacts with some other part of the system by sending or receiving information objects.

```
SCENARIO SQL query exercise

type: usage
repeatedly
    sequence of
        (student, selects, query exercise)
        (student, reads,   query specification)
        (student, submits, query solution)
        (system,  replies, query result)
```

Figure 2. A usage scenario.

A meta scenario – to upload a content unit, here applied to a query exercise – is presented in Fig. 3. This activity contributes to the adaptiveness required to address change and maintenance. Again, this is a composite scenario that refines the corresponding abstract scenario from Table 1.

```
SCENARIO upload content unit (query exercise)

type: meta
repeatedly
    chooses between
        (instructional designer, adds,     query exercise)
        (instructional designer, modifies, query exercise)
```

Figure 3. A meta scenario.

An important feature of our scenario language is the possibility to describe interactions between users and the system and between stakeholders and the software artefacts. Capturing this behaviour is essential for interactive software systems and enables a seamless transition between stages.

6.2 Adaptable architecture design

6.2.1 Architecture and components

Two aspects can predetermine parts of the architecture. Firstly, stakeholder goals and objectives might predetermine architectural decisions; in our case the Web as the most popular teaching and learning platform. Secondly, domain-specific standards often determine the architectural style; for example in

the educational domain the Learning Technology Standard Architecture LTSA [10] determines an abstract component topology. Fig. 4 presents components and connectors for the LTSA. Some interaction processes that would complete an architecture definition with components and connectors will be illustrated in the mappings section.

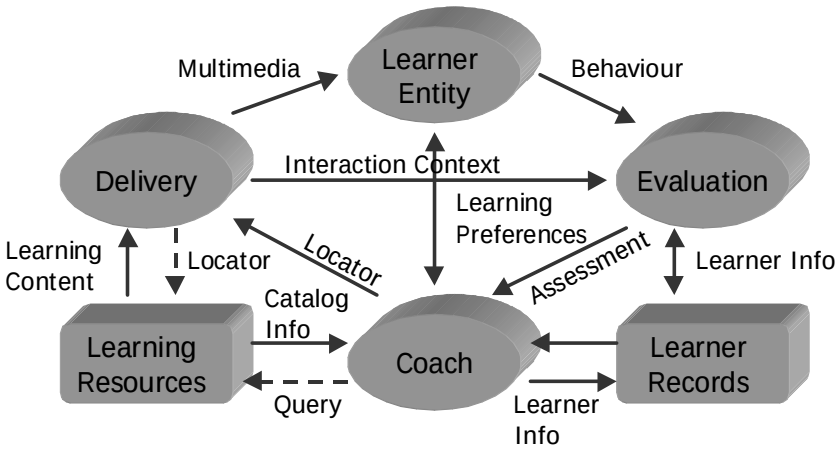

Figure 4. The Learning Technology Standard Architecture LTSA.

We identified the following *subsystems* and *component clusters*, which are in line with the system topology proposed by the LTSA and which also implement a classical three-tiered architecture for Web-based systems (some sample individual components of the clusters are listed in brackets):

- the *backend storage* subsystem with *learning resources* (database material, HTML materials, animation material) and *learner records* (database student) clusters,
- the *server* subsystem with *delivery* (access HTML, access database static, access database dynamic, access animation), *evaluation* (usage mining), and *administration* (registration, login) clusters,
- the *user interface* subsystem with *learner entity* (registration, login) and *instructor entity* clusters.

Essentially, subsystems and clusters are components themselves, composed of more basic components. If the goals and objectives do not indicate an architectural style, the scenarios can determine the full architecture.

6.2.2 Mappings

Architectural aspects such as components and component clusters need to be determined first, then scenarios are mapped onto it; the reflection of the scenarios on the architecture determines prototypes. The scenario-architecture mapping is the central mapping for adaptive development and maintenance here.

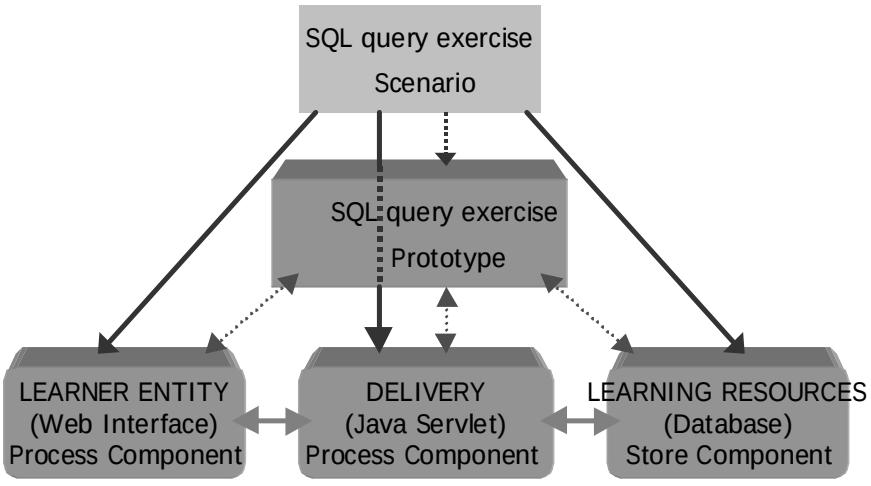

Figure 5. A scenario-architecture mapping.

A sample mapping for the *SQL query exercise* scenario with the associated prototype is presented in Fig. 5. An architecture definition, onto which the scenario and the prototype can be mapped, can be found in Fig. 6. The components are part of the interface, server, and backend component clusters: the learner entity `LE` is derived from the actor `student`, the delivery component `D` is derived from the actor `system`, and the learner resources component `LR` is derived from the `query` objects. The components offer specific services, such as `LE.select`, `LE.submit`, and `D.reply` – these services are fully specified through interface definitions for the components. The connectors – both derived from scenario activities and the LTSA – are `LE <-> D` and `D <-> LR`. The only interaction process we have specified here is the query exercise involving the submission of a solution and the system's reply

```
! ( LE.select(LR.exercise); LE.submit(solution); D.reply(LR.result) )
```

which implements the composite scenario definition (Fig. 2).

```
Architecture

   Components
       learnerEntity     LE (select, submit, ...)
       delivery          D  (reply, ...)
       learnerResources  LR (exercise, result, ...)

   Connectors
       LE <-> D
       D  <-> LR

   Interaction processes
       ! ( LE.select(LR.exercise);
           LE.submit(solution);
           D.reply(LR.result)
         )
```

Figure 6. An architecture definition (simplified).

The important aspect of architectural design is the seamless and traceable transition from the requirements stage that allows us to deal with the initial architecture definition, but also with subsequent changes to the definition.

6.3 Evaluation and evolution of requirements and prototypes

6.3.1 Evaluation of scenarios, prototypes and architectures

Evaluation of scenarios – essentially validating the *completeness* and *correctness*, and analysing the *consistency* – was done in collaboration with users and other stakeholders involved. However, in addition to evaluating text-based representations, we mainly evaluated scenarios through their executable counterparts – *prototypes*, which illustrates their important role in this approach. Scenarios were usually prototyped, i.e. developed in an incremental and user-centred process, if they addressed innovative features with a degree of risk involved. This style of validation through prototypes is suitable for end-user involvement and incremental processes. Prototypes as executable software artefacts allowed us a much richer evaluation addressing a wider variety of quality criteria:

- *Usage evaluation* through data mining and user surveys was used to determine user behaviour and actual interaction processes. Data mining can be used to discover and extract usage processes from Web access logs. General patterns of users' interactions with the system – in the Web environment the users' navigation and response to interactive pages – can be discovered. Usage evaluation based on data mining is critical since it allows us to compare designed (and expected) usage scenarios and their implementation support through interaction processes with actual usage. Web mining based on user activity logs allows constant monitoring and evaluation.
- *Usability evaluation* was based on user surveys and the application of human-computer interaction criteria to the designs. Additionally, other aspects of architectures and prototypes such

as performance had been addressed empirically. Detailed feedback from the users was used to identify functional weaknesses of the prototypes and necessary adaptations.

6.3.2 Evolution

Prototyping started evolutionary cycles of perfective maintenance, i.e. prototyping was the first activity in these phases of adaptive development and maintenance. Prototypes were refined incrementally and have evolved over time. In addition to prototyping, which was mainly a tool for internal and planned evaluations, external and unexpected changes had to be dealt with. Changes that have happened as a result of evolution (volatile and emerging requirements) relate to the system content (information model), the interactions and dialogues (behaviour), the software and hardware technology (infrastructure), and overall system organisation (system features). Both extensions and modifications have taken place.

6.3.3 Analysis and change

Analysis of *scenarios* usually focuses on interference and similarity analyses. *Interference* – referring to sharing of objects (depending on the activity, e.g. write access to information objects) – has caused us no problems. *Similarity analyses*, however, trying to identify the same episodes have resulted in improvements. Similar learning processes for different topics were used to develop generic artefacts. Concrete prototypes (e.g. a relational algebra animator) were generalised into generic scenarios and prototypes and re-instantiated in different forms in an iterative process. The animator prototype was generalised by extracting the dialogue pattern and the interaction channel. A normalisation tool is another example of an application of the generic scenario. Scenarios and prototypes can exhibit reusable structures.

The *maintainability* and *reusability* of the *architecture* can be enhanced through the application of component technology for the architectural aspects. A *cohesion/coupling analysis* of the architecture was carried out; together with the Web and LTSA constraints it has determined the cluster/subsystem organisation of components.

The focus in our case study using the adaptive maintenance approach was on *maintenance prediction*, supported by heuristics and analyses – involving indicators based on evaluation and evolution frequency and severity measures that we have obtained over time for the application domain. We used a *change impact analysis* for the *scenario-architecture mapping* to determine the components affected and investment required resulting from changes in requirements. With little interference, *consistency* was easy to maintain for changes. The meta scenarios have guided the adaptation, i.e. the *change implementation* process, based on *tracing changes* from requirements to the implementation artefacts.

7. Methodology evaluation

Addressing stakeholder needs and change, caused by evaluation and evolution, through adaptive development and maintenance has been our key objective. We have defined several aspects that a methodological framework to improve software productivity in our context has to address:

- *Participation*. The system attributes usability and utility are of paramount importance in interactive systems. Only the constant involvement of the user in the development and maintenance process through formative evaluations of prototypes and adaptation to evolving requirements can achieve the expected quality in terms of these system attributes.
- *Seamless* and *traceable transition*. The transition between different stages and artefacts needs to support the incremental and iterative approach. Smooth transitions need to address the representation of interaction and information access in all representational forms.
- *Adaptive maintenance*. To adapt software and its specifications to evaluation results and evolution is central in environments where change is ubiquitous. Flexible architectures that allow us to control the change impact and analysis technique supporting the change implementation are crucial.

We have introduced an integrated and coherent methodology – combining and adapting existing approaches – that supports the problems arising from change in particular for interactive systems. Being based on a combination of existing, well-known methods and integrated through common principles, has helped us to create a feasible methodology that is easy to understand and use.

We expect a methodology to be effective and to result in improvements in relation to the software product and also in relation to the development and maintenance process.

- *Quality improvement*. Usability and utility were the two central quality criteria of interactive systems reflecting the stakeholder interests. In particular complex interaction processes often require an incremental implementation process. We have constantly measured usability and usage through a range of techniques (usage mining, surveys, etc.) resulting in constant improvements – a result that shows the effectiveness of the methodology in this aspect.
- *Risk minimisation*. Cost-effectiveness and in particular the predictability of costs are crucial in software development projects. In addition to a seamless development process involving stakeholders to ensure the quality of the product, embracing change from the outset is the key to control unexpected maintenance work resulting from evolution and change in the environment. Maintenance prediction and change determination and implementation techniques have helped us to control and manage change, and, therefore, take out some of the risk involved.

The methodology is therefore effective and successful in achieving improvements for the product quality and the process implementation – it focuses on problems typical for the targeted domain of software systems. We have presented a methodological framework here that targets requirements engineering and architectural design aspects; consequently, it needs to be complemented by detailed design and implementation methods.

The methodology itself has evolved over a period of time. In incremental steps we have tested existing and added new techniques and methods, resulting in a demonstration of feasibility and effectiveness. In particular, change-related problems arising from the maintenance of a long-running software system have strongly influenced the methodology. We found hypotheses about software evolution dynamics, as formulated in Lehman's laws [24], confirmed by our experience.

For a long time, the integration of industrial-strength methodology (which has been rather developer-centric) and user-centric approaches (which have often neglected the day-to-day business of development and maintenance) has not been attempted. It is only recently that the need has been recognised and combinations have emerged [2]. Ideally, the solution to such an integration of methods and approaches would be based on a common core aspect that allows both to be combined – change and adaptiveness form this aspect here. A coherent approach is a central requirement. Participative, scenario-based techniques can be integrated with industry-standard methods through behaviour specification and other contractual obligations.

The central lessons that we have learned by being involved in software development and maintenance projects are the following. Firstly, change is ubiquitous and comes in a variety of forms. Secondly, dealing with users (and other stakeholders) is an incremental process. Adapting a system to its environment is the key issue in addressing these two issues. To adapt is the activity that most accurately describes what had to be done from a software engineering perspective – both for the incremental user-centric development and also the iterative, evolutionary process. We have used the presented integrated methodology successfully in several projects covering educational and commerce applications. Evaluations have shown that the systems usability had been increased and the systems maintenance had been made more cost-effective and more amenable to change.

8. Conclusions

User-centricity, or better *stakeholder-centricity*, is essential to address the needs and to meet the expectations of all parties involved in the development and deployment lifecycle of software systems. Users are often essential *drivers* of development and maintenance. *Change* is another essential driver in the lifecycle of a software system. The causes of change can be planned and internal or unexpected and external. Central to both aspects – stakeholders and change – are the *links of a software system with its environment*. Both aspects are an expression of this relationship. The most important requirement concerning software in its environment is the *ability to adapt a system* to the needs of and changes in the environment. Our main observation is that requirements evolution as a consequence of change in the environment has become a central aspect of requirements engineering.

Our *adaptive development and maintenance* is based on participative, architectural, and evolutionary development approaches involving scenarios, architectures, and prototypes. Similar approaches exist.

- *Artefacts*. Besides scenarios, use cases [24] and viewpoints [18] are classical requirements engineering artefacts. In the context of participative design scenarios are the most suitable to focus on activities of stakeholder and on their (partial) execution through prototypes. In order to integrate scenarios and architectures we have extended architectural descriptions by interaction processes. These make the integration with widely used design notations such as UML, e.g. sequence and interaction diagrams, straightforward.
- *Process models*. Similar process models exist. A close example is Boehm's spiral model [24] – equally based on a cyclical model of development including prototypes to minimise risks in the process. However, distinctions between evaluation and evolution are not made; change management is not a concern.

The novel concept behind our approach is adaptiveness, focusing on the relationship between a software system and its environment.

A new technology, like our methodology, has to be seen in the context of likely future developments in area.

- Usability and interactions between users and the system will remain paramount issues – with stakeholder expectations rising. *Interaction processes* will play an important role in software systems design. Interaction is not only a technical term capturing the communication in a computer system; it needs to be addressed at a level reflecting the knowledge and the goals and strategies of the user. Consequently, addressing interaction in a coherent way through the stages and representations is the challenge.
- *Adapting to change* is already the central problem. The relationship between a software system and its environment – managing this relationship is the rationale of our methodology – is an equally important factor in other, related software engineering techniques that will become more and more important in the future. Reusability is an approach to managing and controlling the relationship between a reusable component and its environment. We, therefore, expect reusability and adaptive development and maintenance to benefit from each other in the future.

References

[1] T.A. Alspaugh, A.I. Anton, T. Barnes, B.W. Mott, An Integrated Scenario Management Strategy, Proceedings IEEE International Symposium on Requirements Engineering, 2001, pp. 142-149.

[2] P. Beynon-Davies, S. Holmes, Design breakdown, scenarios, and rapid application development, Information and Software Technology 44 (2002) 579–592.

[3] S. Bødker, Scenarios in user-centred design – setting the stage for reflection and action, Interacting with Computers 13(1) (2000) 61-75.

[4] T. Borgholm, K.H. Madsen, Cooperative Usability Practices, Communications of the ACM 42(5) (1999) 91-97.

[5] K. Breitman, J.C. Sampaio do Prado, Scenario Evolution: A Closer View on Relationships, Proceedings 4th International Conference on Requirements Engineering ICRE'00, IEEE Press, 2000.

[6] I. Crnkovic, M. Larsson, Building Reliable, Component-based Software Systems, Artech House Publishers, Boston, 2002.

[7] A. Dix, I. Finlay, G. Abowd, R. Beale, Human-Computer Interaction, Prentice Hall, 1993.

[8] A. Egyed, N. Medvidovic, C. Gacek, Component-based perspective on software mismatch detection and resolution, IEE Proceedings – Software 147(6) (2000) 225-236.

[9] K. Grønbæk, M. Kyng, P. Mogensen, Toward a Cooperative Experimental System Development Approach, in: M. Kyng, C. Mathiassen (Eds), Computer and Design in Context, 1997, pp. 201-238.

[10] IEEE Learning Technology Standards Committee LTSC, IEEE P1484.1/D8, Draft Standard for Learning Technology – Learning Technology Systems Architecture LTSA, IEEE Computer Society, 2001.

[11] R. Kazman, S.J. Carriere, S.G. Woods, Toward a Discipline of Scenario-based Architectural Evolution, Annals of Software Engineering 9(1-4) (2000) 5-33.

[12] G. Kösters, H.-W. Six, M. Winter, Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications, Requirements Engineering 6(1) (2001) 3-17.

[13] G. Kotonya, I. Sommerville, Requirements Engineering: Processes and Techniques. Wiley & Sons, 1998.

[14] J. Lee, K.-H. Hsu, Modeling software architectures with goals in virtual university environment, Information and Software Technology 44 (2002) 361-380.

[15] C.-H. Lung, S. Bot, K. Kalaichelvan, R. Kazman, An Approach to Software Architecture Analysis for Evolution and Reusability, Proceeding CASCON Conference, 1997, pp. 144-154.

[16] N. Medvidovic, D.S. Rosenblum, R.N. Taylor, A Language and Environment for Architecture-Based Software Development and Evolution, Proceedings International Conference on Software Engineering ICSE'99, 1999, pp. 44-53.

[17] F. Moisiadis, Prioritising Scenario Evolution, Proceedings 4th International Conference on Requirements Engineering ICRE'00, IEEE Press, 2000.

[18] B. Nuseibeh, J. Kramer, A. Finkelstein, A Framework for Expressing the Relationships between Multiple Views in Requirements Specification, IEEE Transactions on Software Engineering 20(10) (1994) 760–774.

[19] C. Pahl, Managing evolution and change in web-based teaching and learning environments, Computers & Education 40(1) (2003) 99-114.

[20] A. Russo, B. Nuseibeh, J. Kramer, Restructuring Requirements Specifications for Managing Inconsistency and Change: A Case Study, Proceedings International Conference on Requirements Engineering ICRE'98, 1998.

[21] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[22] A.F. Smeaton, Using Hypertext for Computer-Based Learning, Computers & Education 17(3) (1991) 173-179.

[23] A.F. Smeaton, G. Keogh, An Analysis of the Use of Virtual Delivery of Undergraduate Lectures, Computers & Education 32(1) (1999) 83-94.

[24] I. Sommerville, Software Engineering, Addison Wesley, 2001.

[25] A.G. Sutcliffe, N.A.M. Maiden, S. Minocha, D. Manuel, Supporting scenario-based requirements engineering, IEEE Transactions on Software Engineering 24(12) (1998) 1072-1088.