Dependency Charts as a Means to Model Inter-Scenario Dependencies

Depicting and Managing Dependencies between Scenarios

Johannes Ryser Martin Glinz

Institut für Informatik
Universität Zürich
Winterthurerstrasse 190
CH-8057 Zürich, Switzerland
[ryser, glinz]@ifi.unizh.ch

Abstract: Scenarios/use cases have gained wide-spread use over the last couple of years. In software engineering they are mainly used to capture requirements and specify a system. Many software engineering approaches, most notably the UML (Unified Modeling Language [RJB99]), use some notion of scenario to support requirements elicitation and to provide a means for improved communication between software engineers, customers and users, and for enhanced user integration in the software development process.
Yet prominent and renowned approaches like the UML lack a concept for modeling dependencies and relations between scenarios and offer only little support for the description and management of scenarios and of inter-scenario relationships.
However, dependencies between scenarios are common, in fact dependencies between scenarios occur in any software development project of reasonable size. The existence of dependencies among scenarios needs to be perceived, acknowledged and accepted, they have to be captured and modeled to fully specify and better understand the system, and their impact on system modeling, on design, implementation and on testing needs to be recognized.
In this paper we argue that dependencies among scenarios play too important a role in the software development process to omit them from system models and not to consciously consider them in analysis, design and testing. Therefore, we introduce a new kind of chart and a notation to model dependencies among scenarios. We discuss briefly the reasons why a new kind of chart is needed. An example of a dependency chart is presented.

1 Introduction

Scenarios are – in the context of requirements and software engineering – sequences of interactions between users and a system. As such, they are mainly used in the analysis phase of software development to elicit, capture and document requirements. However, scenarios are helpful in many other activities as well: manuals need to be written in a user-centered view – scenarios take this view by default and thus are well-suited as a basis for user manuals –, requirements, design and implementation need to be validated by users (again a user-centered view of the system is needed), and, as test cases need to be

developed, scenarios come in handy again, as they are abstract-level test cases and may well be (re-)used for the development of test cases.

For these (and other) reasons scenarios have gained wide-spread use in the RE/SE community. However, current scenario approaches do not exploit many of the potential benefits of scenarios. Partially this is true because scenarios are defined informally only and current approaches do not capture and depict dependencies among scenarios. Some distinct proponents of scenario-based approaches have rooted in favor of not depicting dependencies between scenarios [Ja95, JC95]. The rationale behind this is to keep use case models simple ("Thus, actors and use cases are the only occurrences, phenomena, or objects in a use-case model – no more, no less" [Ja95]). Timing, data, resource and causal dependencies are not to be included in the use-case model. Other dependencies as for example structural dependencies (use case aggregation) are not encouraged because they "support functional decomposition, which would lead easily to a functional rather than object-oriented structure" [JC95]. Thus, in standard modeling languages dependencies among scenarios are hardly ever considered and modeled.

However, scenarios are partial descriptions of system behavior, and as such, they are often applicable to restricted situations only. In most applications, the ordering of scenarios is at least partially not arbitrary and copious dependencies between scenarios exist. In our opinion, dependencies between scenarios carry important information about a system and hence have to be modeled, more especially as relations between and dependencies among scenarios are common and need to be considered if scenarios are to be used to their full potential. Therefore we introduce a notation and diagram to capture and represent these dependencies.

The rest of this paper is structured as follows. In section 2 we introduce the main concepts, the modeling constructs and appertaining notation. A short example illustrates the notation. In section 3 we take a look at related work. Section 4 concludes the paper with a brief discussion of the pros and contras of a new notation and a final argument for the need of a dependency model.

2 Dependency Charts

In the context of this paper, scenarios are defined as descriptions of (possible, future) sequences of interactions between partners, usually between a user and an (existing or imagined) system. Scenarios may be concrete sequences of interaction steps (instance scenario) or they may comprise a set of possible interaction steps (type scenario). The term *type scenario* is equivalent to *use case* in Jacobson's terminology [Ja92, RJB99].

According to definition, scenarios are partial descriptions of system behavior. Furthermore, as has been mentioned before, they often are applicable in restricted situations only. Consequently, scenarios are often not independent of other scenarios. The order and the timing of scenario execution often is not arbitrary. As an example consider the well-known ATM (automated teller machine) system. A scenario description of this system might comprise the scenarios "Withdraw Cash" and "Inquire Balance". Obviously, each one of these scenarios is but capturing a small part of the system's behavior (thus being partial), and the mentioned scenarios may only be executed in the restricted situation of a customer owing a valid card and knowing the correct PIN. So in order to

get a full picture of the whole system, the scenarios either need to be integrated or dependencies between scenarios have to be modeled.

Furthermore, in large systems the scenario model is growing very complex. A structuring mechanism is needed for abstraction and decomposition of complex models into pieces that can be handled more easily.

Current scenario approaches offer but very limited support for abstraction or decomposition, and hardly any support for depicting dependencies. Yet it is important to know the dependencies between scenarios for specifying, designing, implementing and testing the system. Therefore, a language to describe dependencies between scenarios is needed.

These deficiencies in current approaches motivate us to introduce a new diagram type which we call dependency chart. Dependency charts serve some distinct and diverse purposes. On the one hand, dependency charts are used to help the developer gain a clear understanding of the system's high-level dependencies and connections between scenarios, thus supporting the development of more accurate and meaningful models of the system. On the other hand, dependency charts facilitate for a hierarchical decomposition of the scenario model and thus serve as a means to handle complexity. And thirdly, dependency charts may be used to better support testing activities by allowing for reuse of scenarios in testing. In the following sections the concepts and notation used in dependency charts are introduced and a short example is given to illustrate the notation and use of dependency charts.

2.1 The Concepts

Scenarios are dependent on other scenarios in many ways: one scenario needs to be preceded by another one, a scenario might not be run in parallel with another scenario, data needed by scenario A is prepared by scenario B in a producer-consumer-relation, scenario A calls scenario B in its normal flow or to handle an alternative or exception, and so on…

Dependencies between scenarios fall into one of the following categories: abstraction, temporal or causal dependencies. Abstraction dependencies are introduced into the model by hierarchical decomposition of model elements, by aggregation, generalization and refinement structures. Scenarios arranged in hierarchies, scenarios to cover variants (e.g. the same scenario with various slight differences is true for a system depending on hardware configuration), scenarios composed of sub-scenarios and the like, all of them establish abstraction dependencies.

Temporal dependencies establish a sequence dependency between scenarios, e.g. a scenario needs to be preceded or followed by another scenario. Temporal dependencies map to strict sequences (see Figure 2) or to real-time dependencies in dependency charts.

If scenario B may only be executed under certain conditions and scenario A establishes these conditions, then the two are related by a causal dependency. Causal dependencies usually establish a loose sequence (Figure 2): scenario B may be executed any time the conditions hold. Data and resource dependencies are special cases of causal dependencies. An example of a data dependency would be that specific data items need to be created in scenario A before scenario B can be executed. An example of a resource dependency is that an electronic connection needs to be established before scenario B can be performed.

Most temporal and causal dependencies may be captured by execution order. With respect to their execution order, scenarios may be related one to another in four ways: one scenario follows the other one (sequence), either one or the other scenario is executed (alternative), a scenario is executed multiple times (iteration) and a scenario runs concurrently with another one (concurrency). Data and resource dependencies need to indicate what data items and resources are concerned. Abstraction relations may be shown in an abstraction hierarchy.

So we need a model or a notation, respectively, that allows to represent:

- sequences, alternatives, iteration and concurrency,
- general or generic dependencies as for example data and resource dependencies,
- time dependencies that are not captured in sequencing,
- a structuring mechanism like hierarchical decomposition.

Further concepts may be integrated as desired (e.g. frequency of execution of a given scenario as compared to other scenarios to be shown in the model, etc.).

We want to depict dependencies between scenarios in a graph: the nodes shall represent the scenarios, the edges shall represent the dependencies. Sequence, alternative, iteration and parallelism shall be represented in an expressive way. Dependencies of the different types (temporal, causal, abstraction) shall be represented. The notation shall be intuitive.

2.2 The Notation

In dependency charts, scenarios are shown as rectangles with rounded corners and circular connectors attached to both small sides of the rectangle. The connector circles represent the entry and the exit point(s) respectively. Scenarios without any connecting lines (dependency lines) to other scenarios may be executed as many times as desired and in free order, even in parallel with other scenarios if appropriate.

Figure 1: Scenario representation in dependency charts: *a.* a single scenario and *b.* a group of independent scenarios

The (horizontal) position of scenarios indicates the relation of the scenarios one to another. If scenarios may be executed without any restriction on sequence, that is, if they are not required to be in a specific order, then scenarios are drawn in parallel, all the rectangles aligned to the left (Figure 1b). If scenarios are to be executed in a specific order (sequence), they have to be arranged accordingly (see the paragraphs below and Figure 2). The length of the scenario representation does not carry any meaning, in particular it is not indicating duration of execution of the scenarios. If entry or exit points are connected by dependency lines that run perpendicular to the scenarios it indicates simultaneousness, that is, the two scenarios start or end at the same time (Figure 10). Vertical spacing and positioning of scenarios does not carry any meaning.

A sequence of scenarios is shown in dependency charts by attaching the scenario representations one to another (Figure 2), either directly by connecting exit and entry points, or by dependency lines from exit to entry point, thus indicating that the first scenario has to be finished before the second scenario starts execution.

We distinguish between strict and loose sequences. In strict sequences the second scenario must follow the first and the first must be followed by the second. Loose sequences are used to express the fact that if the second scenario is to be executed it must be preceded by the first. Strict sequences usually represent temporal dependencies, loose sequences in turn are used for causal dependencies. In dependency charts strict sequences are shown by connecting scenarios by their circular connectors, loose sequences are represented by connecting two scenarios with a dependency line.

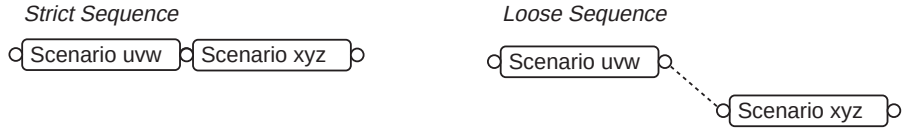

Figure 2: Scenario sequences in dependency charts

An example of a strict sequence is given in the situation of applying for a card for access to a system (e.g. a library card, a bank card to use at an ATM, and so on). The apply-for-card scenario has to be followed by an issue-card scenario in the normal flow of actions. An example of a loose sequence is the scenario of doing statistics on data collected in another scenario: the do-statistics scenario depends on the data-collecting scenario to be executed first, the data-collection scenario however is fully independent from the statistics scenario (it does not have to be followed by the statistics scenario).

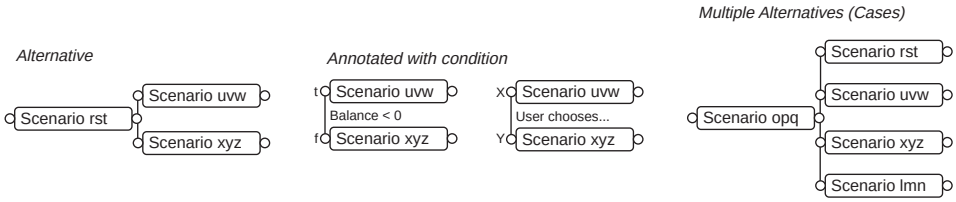

Figure 3: Representation of alternatives in dependency charts

Alternatives are often implicitly specified in dependency charts (usually by scenario name), as any unrestricted scenario may be executed alternatively to other unrestricted scenarios. If alternatives have to be shown explicitly, they are shown by a scenario splitting into two (or more) scenarios. The fork is graphically shown by a perpendicular connecting line from the scenario exit point of the scenario preceding the alternative to the entry points of the scenarios that are executed in dependence of the alternative taken. Alternatives may be annotated with condition(s) or names of the alternatives that can be taken. This is not mandatory, however. Often alternatives are quite obvious in naming of scenarios. Thus, conditions are only specified in dependency charts if needed.

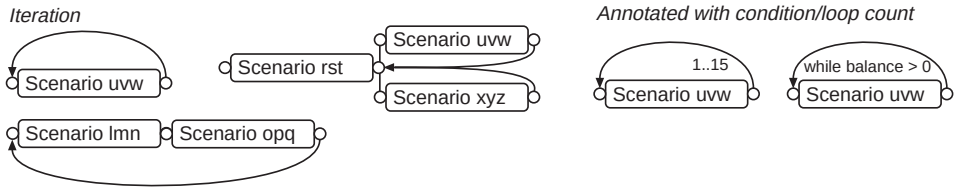

Figure 4: Iterations in dependency charts

Iterations are depicted in dependency charts by backward sloping arrows connecting the appropriate scenario connectors. An iteration may encompass as many scenarios as desired as long as the scenarios are all in the same sequence. An iteration condition may be attached to the arrow-line: Using the well known {0, 1, *} notation to denote multiplicity, an absolute number of iterations may be specified. Other iteration conditions can be expressed using logical expressions and arithmetic (e.g. 'while balance > 0', or 'until number of participants >12').

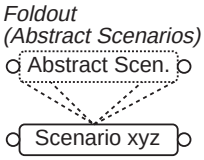

Figure 5: Abstract scenario represented by a foldout

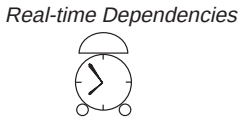

Figure 6: Real-time dependencies

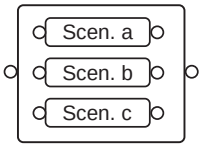

Figure 7: Structuring construct in dependency charts

Real-time dependencies are indicated by the alarm clock symbol (Figure 6). Abstract scenarios (meaning scenario-building blocks that consist of a sequence of actions that are used in more than one scenario, and for this reason are factored out) may be – but do not have to be – depicted as foldouts (Figure 5).

To structure the dependency model we use a hierarchical structure. Scenarios which belong together according to some criterion may be packaged in a box (Figure 7). The chunks that belong together are identified by the dependencies between them. As is the case in modularizing a program, the goal is to keep coupling between packages low and to have cohesive packages. Business processes might define a first set of packages. On the next level down, workflow and dependency structures between scenarios often define the packages in a natural way. If further decomposition is needed, concurrent and repeating blocks in the scenario dependency structure lend themselves for dividing the model into hierarchically decomposed logical parts (for an example see Figure 12, where decomposition blocks are determined by repeating blocks: in the scenario flow, the scenarios that are concerned with the registration and deletion of users and books enclose the scenarios of lending and returning books. The repeating blocks are marked by iteration arrows on the decomposition blocks).

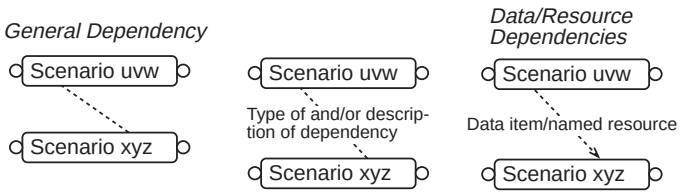

Figure 8: General dependencies

Annotated dependency lines are used to depict any other dependency (general dependencies, data/resource dependencies). Dependency lines should be named or annotated with needed information where appropriate. A dependency line normally is an undirected line and does not specify which scenario is dependent on which other. However, this information often is implicitly given by naming of scenarios. Furthermore, dependency lines may be directed to indicate the dependent scenario. In this case a dashed arrow line is

used to represent the dependency. The scenario at the tail of the arrow depends on the scenario at the arrowhead (Figure 8). If scenarios are mutually dependent this fact may be emphasized by using a double-headed arrow.

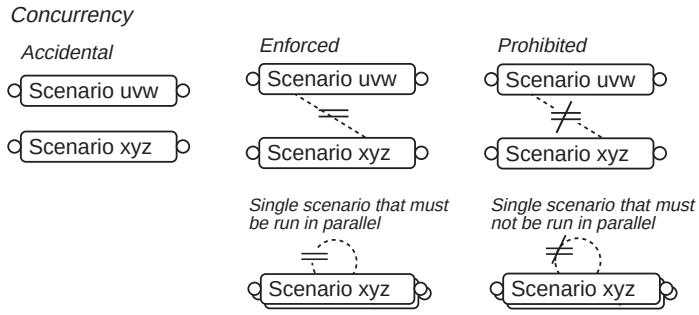

Figure 9: Concurrent scenarios

Concurrency is shown in dependency charts by use of parallel lines (Figure 9). As has been pointed out before, unrestricted scenarios may run concurrently. Therefore, accidental concurrency does not have to be shown explicitly. If concurrency has to be enforced or prevented, the scenarios are connected by "have to be executed in parallel" (=) and "no parallelism" (≠) marks, respectively.

Thus, unbound scenarios (which is the normal case) may, but don't have to, be executed in parallel (concurrency being an accidental feature), scenarios marked with ≠ must not be run in parallel and scenarios marked with = must be executed in parallel (Figure 9).

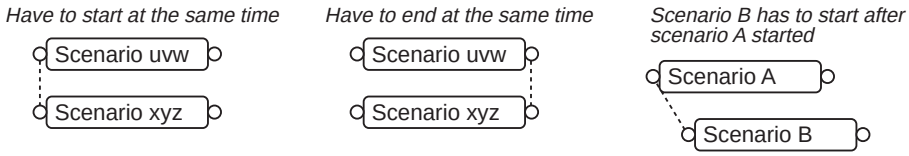

Figure 10: Special cases of concurrency

Many different shades of concurrency of scenarios can – if necessary and desired – be made explicit in dependency charts: Scenarios that have to start at the same time are connected by dashed dependency-lines connecting the entry nodes of the scenarios, the connecting line being rectangular to the baselines of the scenarios. Likewise, scenarios that have to stop at the same time are marked by a dashed dependency line from the exit-point of one scenario to the exit-points of the other scenarios (Figure 10). If scenario A always has to start before scenario B and the two will run concurrently, they are connected by a slanted dependency line from the entry-point of scenario A to the entry-point of scenario B.

A small example is given below to illustrate the representation of scenarios and their relations and dependencies in a dependency chart (Figure 11). A further discussion of dependency charts and more especially of the relationship to other modeling languages may be found in [RG00].

2.3 An Example

As an example illustrating the creation and use of dependency charts, we choose the well-known library example. In a library, the user can apply for a library card to be allowed to borrow books and search for books. In the example, there are two actors: the library user (borrower of books) and the librarian. There are five scenarios in the example in which the library user is the actor: (01)Apply for library card, (02)Query catalog, (03)Borrow books, (04)Return books and (05)Apply for deletion from user catalog. The scenarios for the actor 'librarian' are: (11)Register user, (12)Delete user, (13)Update user data, (14)Catalog book, (15)Remove book, (16)Maintain library catalog, (17)Query user data and status, (18)Query book status, and (19)Call overdue books. In a real system, there certainly would be some more scenarios (and thus most probably some more dependencies…), but in the paper, to keep the example short, we limit the system to the scenarios listed above.

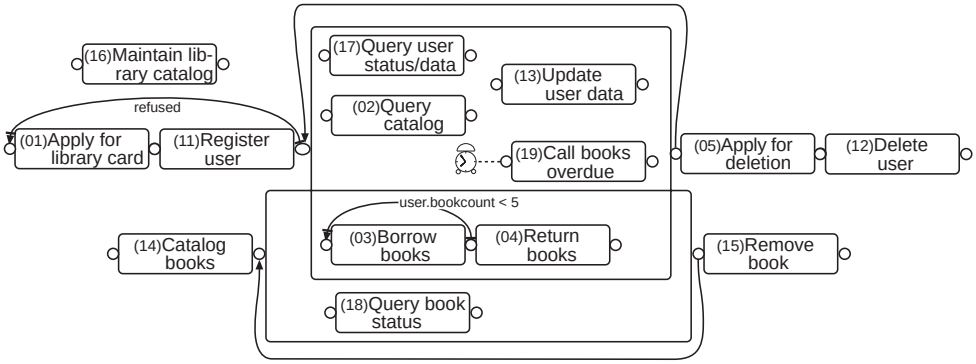

Figure 11: An example dependency chart

Some of the sequence dependencies are quite obvious (as readers are familiar with the domain and know the situation in and the context of a library very well): first a potential library user has to apply for a card, then the librarian has to register the new user, before the library may be used as many times as desired by the – now regular – user. The library user queries the catalog and borrows books up to the limit of four books per user at a time. Books have to be cataloged, before they can be borrowed. Finally, the borrowed books have to be returned in time (else an overdue note will be sent by the librarian) before they can be borrowed by another user.

The librarian on the other hand maintains the library catalog. She may query a user's status and update a user's data, once the user is registered. If a request for deletion from the user catalog is sent, the librarian will delete the user, after having called back all the borrowed books from this user. Furthermore, she will catalog new books and remove stolen, old and torn ones. A book first has to be cataloged before its status can be queried or before it can be removed.

In dependency charts we do not enforce a strict hierarchical structure. If desired, hierarchically-structured model elements may overlap as illustrated in Figure 11. We allow these overlapping elements to support aspect-oriented modeling in an integrated model. Thus, the fact that a scenario is dependent on multiple entities or actors can be modeled explicitly in dependency charts.

In the example (Figure 11) the scenario of a book being borrowed depends on the book being catalogued *and* a borrower being registered (so that she may borrow the book). This is expressed by including the '*Borrow books*' scenario in both, in the repeating block of the book and in the repeating block of the user life cycle.

3 Related Work

Dependency charts have some properties that closely relate to other graphical models or modeling languages. The abstraction mechanism is quite similar to hierarchical decomposition in structured analysis. Concurrency, synchronization and non-determinism are similar to Petri-nets. However, contrary to Petri-nets, dependency charts depict mainly the statics and less the dynamics of relations between scenarios. Sequences, alternatives and iterations are part of all notations that are used to model control-, data- or workflow. Thus, dependency charts are much like flow-charts representing the structure of control flow between scenarios. They do include some further information, though, that is missing in true flow-charts (e.g. time dependencies, data and resource dependencies, parallelism, …). The problem of loss of structure that is a concern in flow-charts is of equal import in dependency charts as well: any connection in-between scenarios is possible, even if the depicted flow does not make any sense in reality. Developers need to be aware of this and model carefully.

In most popular modeling languages, notably in the UML also, there is no predefined way of how scenarios are to be represented and modeled. The only diagram specific to scenario modeling is the use case diagram [RJB99, Ja92] depicting the associations between actors and use cases. The details of scenarios may be modeled using natural language descriptions, interaction diagrams, activity charts, statecharts or some other form of representation. Mostly it will be natural language descriptions. An integrated scenario model does not exist and is not intended.

Only three types of inter-scenario relationships can be modeled in UML use case diagrams: Generalization, «Include» and «Extend». Generalization relates general use cases to special case use cases, providing some means of abstraction. «Include» means that the behavior of the included use case is part of the including use case as well (which corresponds to a procedure call in programming). «Extend» means that the extending use case is inserted into the extended one at a designated extension point if a guarding condition is true (a mechanism corresponding to macro expansion in assembler programming).

Expressing sequential, parallel or iterative relationships between use cases as well as time and data dependencies is impossible in UML use case diagrams. Neither is there a systematic way for decomposing use cases in UML [Gl00b]. Thus most dependencies between scenarios can either not be represented at all in UML or they have to be expressed in terms of preconditions and annotations to the detailed scenario descriptions. However, modeling dependencies with preconditions and annotations makes them difficult to recognize and to trace.

Clearly, the notation of use case diagrams could be extended to include the necessary structures to depict general dependencies, but the original intention of the diagram would be lost in doing so. Therefore, we think it is advisable to use a distinct notation to emphasize the difference of the new diagram from existing ones in meaning and use.

Glinz [Gl95, Gl00a] has proposed statecharts and Jackson-style diagrams for expressing decomposition and structure of scenarios. This is a closely related approach. However, it is less general than dependency charts are, as it represents sequence, alternative, iteration, concurrency and composition only.

4 Conclusions

In this paper we have introduced a new diagram type for modeling dependencies between scenarios. We have argued that dependencies among scenarios are common as scenarios are partial models that apply to restricted situations only. Therefore, it is important to know, understand, document and manage dependencies to build an accurate system model. Dependencies need to be known in all system modeling activities, as well as in testing. For example, we need them for a precise and correct system specification, for an accurate design and for thorough testing. Moreover, a model of dependencies between scenarios greatly enhances the understanding of the system to be built. Questions like "What other requirements, scenarios, … will be affected if this requirement, scenario, … is changed?" can be answered because dependencies between the different scenarios are known. Thus, traceability is enhanced, design alternatives can better be evaluated, and effort and cost estimation can be improved. Maintenance profits greatly if a dependency model is available. Furthermore, in testing the tester needs to know the dependencies else he will not be able to develop test suites from a scenario model that test the relations, connections and dependencies between different use cases.

Finally, the notation presented in this paper features a construct to decompose large models. These are sorely needed if working with large systems and models, but are missing in current modeling languages such as UML.

In summary we argue that a distinct model to capture dependencies pays the effort of creating and maintaining it as it greatly enhances the scenario model.

References

[Gl95] Glinz, M.: An Integrated Formal Method of Scenarios Based on Statecharts. In (Schäfer, W.; Botella, P. eds.): *Software Engineering – ESEC '95. Proceedings of the 5th European Software Engineering Conference*, Springer, Berlin, 1995; pp. 254-271.

[Gl00a] Glinz, M.: Improving the Quality of Requirements with Scenarios. In: *Proceedings of the Second World Congress on Software Quality*, Yokohama, 2000; pp. 55-60.

[Gl00b] Glinz, M.: Problems and Deficiencies of UML as a Requirements Specification Language. In: *Proceedings of the Tenth International Workshop on Software Specification and Design*, San Diego, 2000; pp. 11-22.

[Ja92] Jacobson, I. ; Christerson, M.; Jonsson, P.; Övergaard, G.: *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Amsterdam, 1992.

[Ja95] Jacobson, I.: Formalizing Use-Case Modeling. *Journal of Object-Oriented Programming*, vol. 8, # 3, 1995; pp. 10-14.

[JC95] Jacobson, I.; Christerson, M.: A Growing Consensus on Use Cases. *Journal of Object-Oriented Programming*, vol. 8, # 1, 1995; pp. 15-19.

[RG00] Ryser, J.; Glinz, M.: *SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test*. Berichte des Instituts für Informatik, 2000/03, Universität Zürich, Institut für Informatik, Zürich, 2000.

[RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.