Innovations Syst Softw Eng (2011) 7:53–69
DOI 10.1007/s11334-011-0144-5

ORIGINAL PAPER

A study to support agile methods more effectively through traceability

Angelina Espinoza · Juan Garbajosa

Received: 8 September 2009 / Accepted: 19 January 2011 / Published online: 19 February 2011

Abstract Traceability is recognized to be important for supporting agile development processes. However, after analyzing many of the existing traceability approaches it can be concluded that they strongly depend on traditional development process characteristics. Within this paper it is justified that this is a drawback to support adequately agile processes. As it is discussed, some concepts do not have the same semantics for traditional and agile methodologies. This paper proposes three features that traceability models should support to be less dependent on a specific development process: (1) user-definable traceability links, (2) roles, and (3) linkage rules. To present how these features can be applied, an emerging traceability metamodel (TmM) will be used within this paper. TmM supports the definition of traceability methodologies adapted to the needs of each project. As it is shown, after introducing these three features into traceability models, two main advantages are obtained: 1) the support they can provide to agile process stakeholders is significantly more extensive, and 2) it will be possible to achieve a higher degree of automation. In this sense it will be feasible to have a methodical trace acquisition and maintenance process adapted to agile processes.

Keywords Traceability methodology · Metamodeling · Agile methods · Test-Driven Development (TDD) · Storytest-Driven Development (SDD) · ISO-24744:2007 SEMDM

A. Espinoza (✉) · J. Garbajosa
Technical University of Madrid (Universidad Politecnica de Madrid - UPM), Systems and Software Technologies Group, E.U. Informatica, Ctra. de Valencia Km. 7, 28031 Madrid, Spain
e-mail: aespinoza@syst.eui.upm.es
URL: http://syst.eui.upm.es

J. Garbajosa
e-mail: jgs@eui.upm.es

1 Introduction

Methodologies such as XP [9] or Scrum [65], that can be included into the agile group [1,32], aim to deal with evolving requirements in a time-constraint scenario using a number of techniques. Among these, we can find Test-Driven Development (TDD) [8] and Storytest-Driven Development (SDD) [55]. In everyday software development, requirements tend to evolve quickly and towards obsolescence even before project completion because of rapid changes in stakeholder preferences, technology, competitive threats, and time-to-market pressures [53].

Agile development processes have a different perspective, compared to traditional development processes which follow a more lineal or waterfall model for performing tasks. This is the case of the requirements engineering (RE) processes, as is pointed out in [12]. One of the differences is that a detailed requirements specification may be missing during a large part of the project or even the whole project duration [72]. Agile approaches such as TDD or SDD advocate the development of code without waiting for formal requirements analysis and design phases. Requirements emerge throughout the development process [12]. Some other differences include the use of stories as a source for requirements. Stories include many details and may be more ambiguous than the conventional requirements specification. A story may also be more coarse-grained than the traditional requirements specification. They are sometimes used together with *user tests*. *User tests*, to some extent, become the fine-grained requirements. TDD considers writing tests as part of the requirements/design activity, in which a test specifies the code's behavior. In practice, and according to [52], many organizations use tests to capture complete requirements and design documentation, that are linked directly to code production. This is an uncommon scenario in a traditional development life cycle, in which

requirements specification is prior to the system development.

Traceability practices in traditional requirements engineering advise setting up traces from requirements to other development artifacts; and requirements are part of a formally structured requirements specification [21,36]. The standard ISO/IEC 12207:2008 [41], in bullet 7.1.5.3.1.5, explicitly specifies that the implementer shall evaluate software code and test results, considering traceability, to software requirements and design items. In fact, ISO/IEC 12207:2008 considers that the System Requirements Analysis has the System Requirements Specification as outcome, which is used in the System Qualification Testing Process to ensure that the implementation of each system requirement is tested for compliance [41, 6.4.6]. A similar scenario is stated in SWEBOK [19, chapter 2], the Requirements Specification knowledge area (KA) describes that a Software Requirements Specification has to be produced, and states that such specification is to be performed before design begins. In SWEBOK, the Requirements Validation KA states that "the requirements may be validated to ensure that the software engineer has understood the requirements, and it is also important to verify that a requirements document conforms to company standards, and that it is understandable, consistent, and complete" [19, chapter 2, Sect. 6]. That is, a requirements specification document is strongly demanded by the software engineering community, as in the case of SEWBOK and ISO/IEC 12207:2008.

However, in an agile approach, requirements traceability (RT) cannot be performed as it is established in traditional requirements management, as ISO/IEC 12207:2008 or SWEBOK demand. This is mainly because a formal system/user requirements specification document is frequently omitted. This situation, if improperly managed, can create serious problems to other processes such as change management, impact analysis, and estimation, which are based on a life cycle model that starts from a traditional requirements analysis process. Some authors think that a missing requirements specification might result in severe problems [56]. As a response to this, several practices should be implemented together with agile requirements approaches to address the lack of a detailed requirements specification [12]. These include establishing a strong traceability practice, together with using explicit requirements negotiation, cooperative strategies for requirements engineering, and incorporating aspect-oriented concepts [12].

Though agile methods do not require to develop a formal requirements specification document, systems quality and dependability are, for sure, a key concern. Traceability is explicitly considered in agile methodologies as a fundamental issue to develop quality systems on time ([64], Sect.1; [57]). A strong reason is that traceability is an excellent support for accountability. Therefore, it is fundamental to develop traceability approaches which tackle the complex traceability requirements stated by agile methods. This paper focuses on studying which traceability models can better support agile development processes, particularly for those methods which implement agile requirements.

Another key issue regarding traceability is link semantics. Link semantics can also be different for agile development methods compared to traditional development processes. Before analyzing links semantics, let us try to understand how some of the agile practices happen. XP commonly uses the planning game. It puts a special focus on the requirements negotiation and, likewise, on the their implementation planning; all the members of the project team meet and discuss the requirements captured by the customer, in the so-called user stories. Tests from user stories act as requirements, and their execution can be regarded as a requirements engineering practice, a design practice and a test practice. Traceability links must have a precise semantics to support this complex scenario. If a requirement is linked to an acceptance test, its semantics from the traditional perspective is clear. However, the semantics is different if a requirement is linked to a user story test, due to the several facets it has. Moreover, in the context of a large project some subsystems may be developed according to a traditional model and some following an agile model. Tests may have a different meaning depending on the lifecycle approach. If accurate link semantics is missing, confusion may arise.

This paper studies some agile development processes and methodologies, such as XP, TDD, and SDD, from a traceability point of view. As a result of this study some requirements for traceability models are obtained. These requirements are needed so that traceability models can support agile development processes effectively. Some existing and well-documented traceability models will be analyzed from the perspective of these requirements. As it will be next discussed, conventional traceability models are not able to provide effective support to agile development processes. These requirements are adequately considered in an emerging traceability model called traceability metamodel (TmM) that is briefly described in Sect. 4. This model has been developed according to proper modeling principles described in [27], and it is being validated at present. One case study is on agile development that is presented in Sect. 5, to show how the proposed ideas work in practice. In previous works [29,30], fundamental issues to define a project-specific traceability methodology were presented and discussed. Some of those results were to include, as part of the traceability implementation, traceability types with linkage rules, support for different granularity levels of the linking system artifacts, and traceability weights to indicate the relationship dependency force between two artifacts. All these concepts are considered in the TmM definition.

This paper is organized as follows: after Sect. 1, Sect. 2 states basic traceability concepts and studies agile processes from the traceability perspective. Section 3 presents some basic traceability requirements to effectively support agile processes; limitations of existing approaches with respect to these requirements are discussed. Section 4 briefly describes the TmM foundations, design criteria and the parts of the metamodel, which are necessary to understand the rest of the paper. Section 5 presents a case study to illustrate the concepts already introduced with an example; a discussion on the advantages and disadvantages of the approach is presented as well. Section 6 analyzes some alternative approaches to the problem under study. Finally, Sect. 7 states the conclusions and future work.

2 Background

2.1 An overview of traceability

A trace is defined in the IEEE Standard Glossary of Software Engineering Terminology [40] as "a relationship between two or more products of the development process", and traceability as

1. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor–successor or master-subordinate relationship to one another.
2. The degree to which each element in a software development product establishes its reason for existing.

The most-referenced traceability definition in literature is provided by Gotel and Finkelstein in [36, pp. 4] as the ability to describe and follow the life of a requirement, in both forward and backward direction, ideally through the whole system life cycle (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases).

Traceability provides an essential support to produce trustworthy and high-quality software systems, as in the case of embedded systems with critical components. Verification and validation (V&V), release management or change impact analysis play an important role to achieve the required level of quality. The relevance of traceability for these areas has already been outlined in [21, pp. 105–109]. Traceability is also fundamental to implement model driven architecture and, closely related, round trip-engineering [11].

Egyed and Grunbacher mention in [25] some important goals of requirements traceability which are: to facilitate communication, to support integration of changes, to preserve design knowledge, to assure quality and to prevent misunderstandings. RT is also crucial to establish and maintain consistency between heterogeneous models used throughout the system development lifecycle.

Numerous techniques have been used for providing RT, and they differ in the quantity and diversity of information they can trace, the number of interconnections they can control between information, and to the extent to which they can maintain RT when faced with changes to requirements [36]. Some examples are as follows:

- Cross referencing schemes [31]
- RT matrices [44,59]
- Graph-based representation [60]
- Keyphrase dependencies [43]
- Hypertext [45]
- Integration documents [47]

Many commercial tools and research products support RT, primarily because they embody manual or automated forms of the above techniques. Some examples of research tools are detailed in [14,16,36,44,51,58]. The commercial tools most used are IBM Rational Rose XDE (for model-driven engineering users), IBM Rational RequisitePro and IBM Telelogic DOORS [5, pp. 4].

Different traceability models such as [4,7,33,50,54,61,68,69] are available in literature. However, a number of challenges such as link semantics, traceability methods and tools, traceability process models, cost-benefit analysis, or measurement and benchmarking, are still to be effectively tackled. The "Center of Excellence for Traceability" in the document *Problem Statements and Grand Challenges in Traceability (COET-GCT-06-01)* [6], summarizes all the remaining challenges.

2.2 Agile versus traditional from the traceability perspective

The differences between agile and traditional development process models (such as waterfall or lineal models) are not only in the way processes are performed, but also in the artifacts produced as process outputs. TDD and SDD consider writing tests as part of a requirements/design activity, in which a test specifies the code behavior. In such situations, the traceability activity requires linking test to code, and not the opposite, indicating that the code realizes and/or satisfies a test, which also acts as a user or system requirement. As a consequence, many of the existing traceability models are able to provide very limited support to some agile development processes, as TDD and SDD. See Fig. 1 which shows the TDD and SDD life cycle model for developing systems. Here the Requirements Analysis stage is performed during the Testing stage: test acts as the system requirements. In more traditional development models as the waterfall life cycle, the Requirements Specification Document is fundamental to start the design and code of the underlying system.

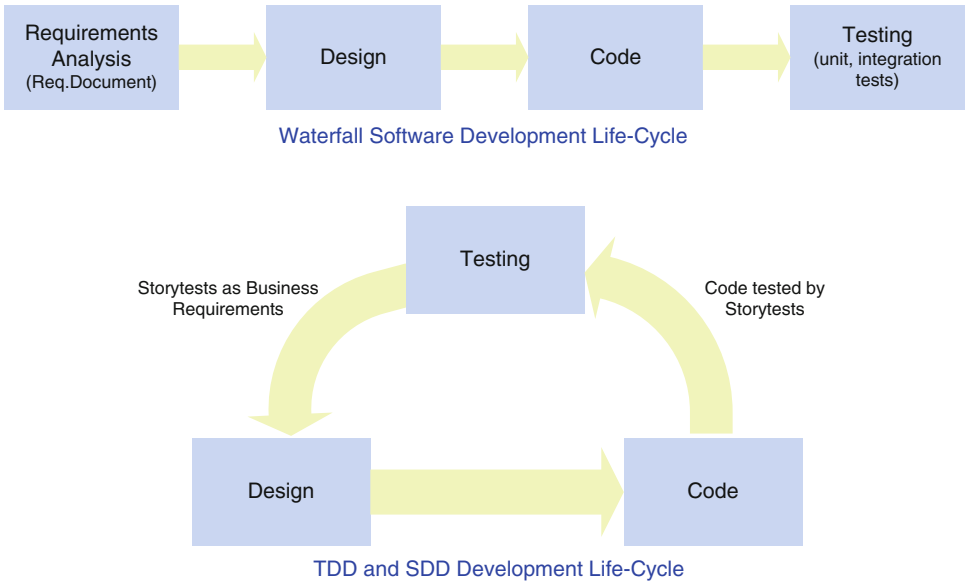

Fig. 1 TDD and SDD development life cycle model, compared to a traditional life cycle model

An issue to take into account is, that according to [17] every project needs its own personalized methodology, based on non-technology characteristics such as team size, geographic separation, project criticality, and project priorities. Many of the available traceability models, as it will be discussed in this paper, are monolithic and their adaptation to individual project needs is very limited.

Following [12] it is strongly recommended to implement a traceability practice in agile projects to prevent problems due to the lack of a requirements specification. However, current traceability approaches are based precisely on a complete requirements specification document. As an example, the acquisition of traces between requirements and design as in [10,13,22,63], or the automation of traceability practices as in [15,16,23,48]. Thus, the need to provide a new and specific approach for traceability in agile processes seems to be a fundamental issue. Moreover, traceability models must be able to support the acquisition of links between artifacts, for which links semantics should be adapted to support an agile rather than a traditional development. As it was mentioned in Sect. 1, a link from a requirement to an acceptance test does not relate the same type of artifacts, than a link from a user story to a test.

Another issue to consider is *granularity*. A user story has a non-formally defined structure. It might happen that, eventually, a user story could be mapped into several requirements. Therefore a lower granularity items such as tests, become essential to support accountability of development decisions. This creates a new situation with respect to traditional processes.

Additionally, another issue to consider is regarding the security over the traceability information. Traditional development processes are performed basically by software engineers. Agile processes are performed by practitioners that are not only software engineers. In fact, user story tests may be developed, executed and assessed by customers with business expertise as described in [55, pp. 3] and [12, pp. 3]. This creates a number of new situations that have to do with process performance and security. Roles have to be a fundamental part of the traceability implementation.

A conclusion of this analysis, pointed out in [29,30], is that traceability support for agile development practices can be improved taking into account at least three modeling issues. The first is that the traceability models must support user-definable traceability types. Link semantics must function as a potential tool to strongly support the development tasks that extensively use traceability data. Methodologies must change according to new challenges and technology changes, as [18] indicates; traceability models and practices, that are part of this methodology, accordingly. The second issue is user-definable roles. Role information is more relevant in agile processes than in traditional processes, due to some stakeholders perform uncommon tasks and they must be executed following a strict and secure plan. The third issue is user-definable linkage rules. This concerns the automation topic: links must be created automatically whenever possible. This is essential for agility. Therefore user-definable linkage rules have to be introduced as a powerful mechanism to improve automation and agility. User-definable linkage rules can be supported by an improved semantics of the link types, and roles.

3 Traceability requirements and agile processes

3.1 Traceability types for agile practices

A great majority of the existing traceability models and methodologies, such as [10,13,38,49,63], assume that one of

the outputs of the development process is a formally structured requirements specification. Even more, the requirements specification would be an early output of the process. Current traceability approaches do not consider that requirements may be expressed in terms of user stories or tests, as it is used in agile approaches such as TDD and SDD. Therefore, existing traceability approaches do not contemplate tracing a test *acting* as a requirement, or a user story to a code item. One way to cope with this fact is that the traceability model, whichever it is, supports user-definable traceability link types, instead of providing merely a set of pre-defined types. Link types would be defined according to project needs. The traceability model should be capable of being *customized*, since each project will have its own characteristics according to [17].

To be really useful for the agile software development process, traceability types must be able to relate any artifact independently of the structure type and granularity. It must be also considered that requirements artifacts could be subjected to iterations, and described in an unusual way from the point of view of the traditional development approaches. That is the case of the FIT language [20] used by customers in TDD and SDD, to define tests that play the role of user requirements. Traceability types must be featured by a strong semantics; this way, testing techniques may become more accurate, reliable and faster.

To be both useful and usable, a traceability system relies, to a great extent, on a methodical trace acquisition and maintenance process. Trace acquisition and maintenance are also essential for getting a good level of automation. User-definable traceability types, as opposed to merely pre-defined types, greatly facilitate methodical trace acquisition and maintenance processes, because of the strong semantics they provide. At present, some approaches manage to automate a large number of traceability tasks, but they do not solve the issue under discussion, since they use mainly natural language processing, or information retrieval techniques, as in [13,15,25,24,66,70,71].

3.2 Traceability roles for agile practices

How stakeholders are involved in agile processes is not the same as in traditional processes; end user involvement is definitely greater in agile. For instance, in SDD the on-site stakeholders may *act* as a test developers and assessors.

It is the on-site customer who is responsible for defining and checking the acceptance tests for each user story, for example with FIT [37,52,55]. Based on this situation, the customer selects the user stories and allocates them to the right releases. This is an uncommon scenario in traditional development processes.

Agile processes will also require increasing security issues on trace information with respect to traditional development processes. This is because some roles, not necessarily technically skilled, will be performing activities that are traditionally assigned to technically skilled roles. For instance, during a release planning meeting the on-site customer may be supported by all success-critical stakeholders from the customer side to refine the high-level requirements developed in the project planning meeting into lower granularity and more measurable statements [37]. The on-site customer re-checks the completeness, consistency, and reliability of this document. These activities are traditionally assigned to the analyst role. In TDD or SDD, the system analyst is frequently eliminated as an explicit role, and the activities performed by the analyst are spread across other roles, such as programmers and testers. Thus, since several non-technical skilled stakeholders may access and modify the traceability information, it is fundamental to monitor it, and not only to monitor the system artifact changes, as usually happens in configuration management strategies.

Therefore, for agile scenarios, traceability models must support the definition of user-definable roles. Each role will be able to perform system development activities adapted for agile practices, such as the on-site stakeholder performing development activities.

3.3 Linkage rules for automation

Traceability is considered an important tool to build high-quality systems using agile development processes [12]. However, although some techniques for generating and validating requirements traceability are available, in practice, creating and maintaining traces are often labour-intensive and complex. Lago et al. in [46] assure that it is very expensive to maintain accurate traceability information similarly to the more general problem of software documentation. If traceability links are kept manually, they are simply not updated or just forgotten as soon as the development deadline approaches. This results in incomplete trace information that cannot assist engineers with real-world problems [25]. This topic is fundamental in agile processes for which it is important to reduce effort during the traces acquisition and maintenance process to empower agile software development.

There are several approaches to automate traceability link acquisition. To mention some, the authors of [67] demonstrate the ability to automate traceability relations generation at reasonable levels of recall and precision. In [13] syntactic analysis of text documents written in natural language is used to manage traceability links; these links are between the initial software requirements and the formal object representations resulting from the modeling processes. Reference [25] focuses on automatically generating dependency links between requirements, design artifacts, test cases and source code. These approaches offer an excellent level of

automation; however, they propose techniques to generate links between pre-defined development stages. As a result, a traditional life cycle model with traditional software outputs is assumed, and this is not the proper scenario for applying agile development processes. As it was discussed in Sect. 2.2, in TDD or SDD early writing of acceptance tests works actually as a requirements-engineering technique [52]. It is the opposite to what happens in a traditional life cycle, where requirements specifications are clearly different from tests as described in ISO/IEC 12207:2008 [41], and discussed in Sect. 1.

Current link acquisition approaches, then, have not been devised to support linking an artifact that can be considered as a requirements specification, and at the same time a test. Therefore, the concept of "user-definable linkage rule" appears to be a fundamental part of a traceability model. The concept specifies the rule or logics to create links between an origin (source) artifact type and a destination (target) artifact type. A lack of linkage rules will prevent to achieve the full automation level in traceability model implementation, deployment and management. On the contrary, linkage rules will facilitate the automation of traceability tasks, particularly link generation and maintenance, as in agile methods.

4 An overview of TmM metamodel for traceability methodologies definition

Section 3 *Traceability Requirements and Agile Processes*, identifies three issues where improvements in traceability models are required to support agile processes effectively: user-definable traceability types, user-definable roles, and user-definable linkage rules. They will *boost agility* through trace acquisition.

This section introduces a traceability metamodel called TmM (Traceability metamodel for methodology definition), introduced in [27], from which it is possible to develop project-specific traceability methodologies. TmM supports the three issues mentioned.

Next subsections introduce briefly the modeling principles used to define TmM. The objective is to provide the necessary background so that the reader can understand that, using the TmM modeling features, this traceability metamodel can widely provide a solution approach to the traceability requirements described in Sect. 3. The advantages of modeling these aspects explicitly, compared to other existing approaches which do not provide customized traceability types, roles and support for linkage rules, are discussed as well.

4.1 TmM extended from SEMDM

TmM has been extended from the Software Engineering Metamodel for Development Methodologies (*SEMDM*)

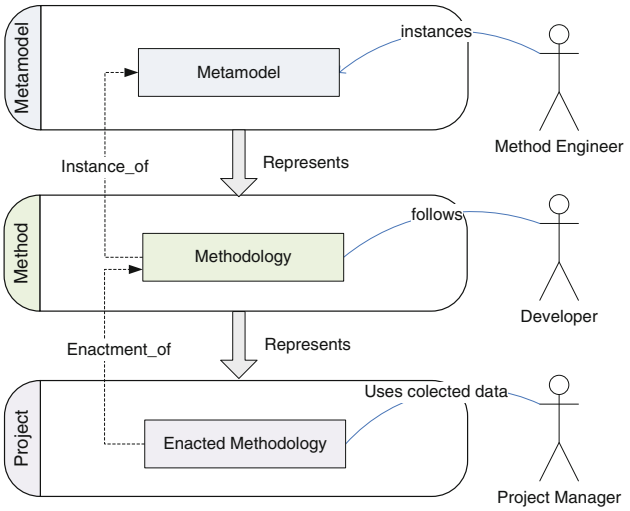

Fig. 2 Three modeling abstraction levels corresponding to the metamodel applicability to three expertise domains (based on [39,34, Fig. 1])

defined in ISO/IEC 24744:2007 [42]. SEMDM is a comprehensive metamodel for defining methodologies and it has been designed to support all kind of methodological concepts and it is independent from a specific life cycle or method. Hence, this feature assures that TmM can widely support agile methodologies. The *ISO-24744:2007* metamodel is already backed on the MOF and UML architecture and notation, which warranties standardization in the metamodeling process which was followed to define the TmM metamodel (see [34], Sects. 2 and 4).

SEMDM makes use of a new approach to define methodologies based on three-layer modelling hierarchy and the concept of powertype patterns. Figure 2 shows the three-layer modeling hierarchy managed in SEMDM: *metamodel* layer, in which the methodology concepts are defined; *methodology* layer, in which a methodology is adapted to the project needs, and the *project* layer, in which the instantiated methodology is used for a given project. A detailed definition of these three abstraction levels for methodologies can be found in [34,39].

The three-layer modeling hierarchy supports modeling traceability concepts, at the three previous expertise levels. Power-type patterns principle is the tool which makes possible to model these traceability concepts through the three-layer modeling hierarchy. Actually, SEMDM provides methodology concepts as powertype patterns and classes.

Powertype patterns is a modelling technique [34,35], to represent a concept in a model with a duality. For instance, to model the test case concept, a powertype pattern defines the *TestCaseKind* class which is used to define all the test cases types, which can be defined in a software development project, such as unit test cases or system test cases. Also, the pattern defines the *TestCase* class which is used to define the

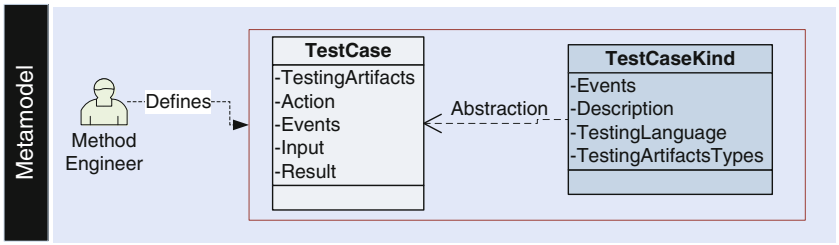

Fig. 3 TestCase powertype pattern

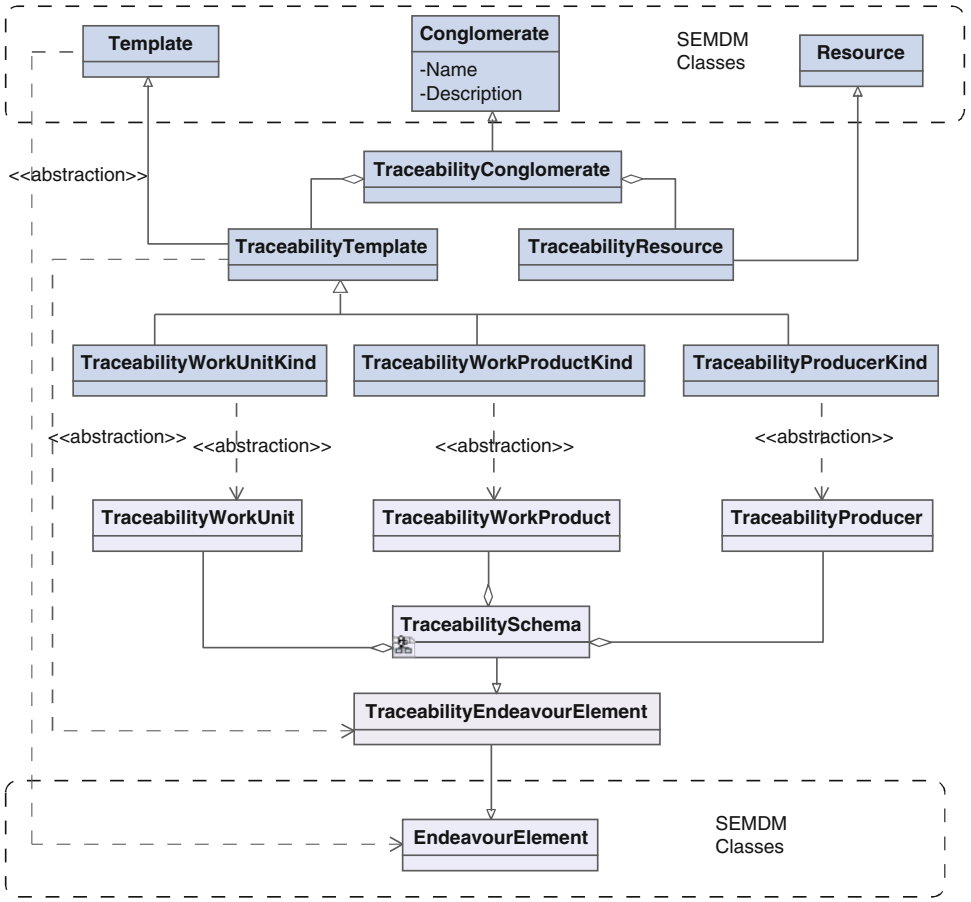

Fig. 4 Traceability methodology, represented with the *TraceabilityConglomerate* class

specific test cases that are used in the project. Figure 3 shows the *TestCase* powertype pattern.

Hence, each TmM diagram will contain powertype patterns and classes, and diagrams will follow the same graphical conventions than SEMDM defines. Classes in the top correspond to the methodology definition, and classes in the bottom correspond to the project development. In the diagrams those classes that are related with the *abstraction* relationship are actually, the powertype patterns of this paper's traceability proposal. The other classes that do not have the *abstraction* relationship, are typical classes of a class diagram.

The instantiation process which describes how to use top and bottom classes through the layers, to generate project-specific traceability methodologies, is detailed in Sect. 4.4.

According to the SEMDM extension rules, the *Conglomerate* class which "is a collection of related methodology elements that can be reused in different methodological contexts" [42] p. 19. Then, the TmM root class is *TraceabilityConglomerate*, corresponding to all the items of a traceability methodology (see Fig. 4). All TmM classes are actually derived from a SEMDM's classes, to be compliant with the standard. This can be consulted in [26], but omitted here for space reasons.

4.2 Traceability metamodel

The TmM metamodel provides the baseline for the systematic and formal definition of a project-specific traceability methodology, and includes three aspects: the process

to follow, the artifacts to use and produce, and the people involved [27]. TmM covers the whole process lifecycle, and does not enforce any process model, either classical or agile.

TmM includes a core set of traceability items, from which a project-specific traceability methodology can be defined. This core set consists of traceability project *templates*, and *resources*. Templates are items that once they have been defined in the methodology, they must be instantiated in the project development, such as traceability types which must be instantiated to create links. Resources are items that once they are defined in the methodology, they are used exactly as their definition, such as traceability metrics. In TmM the templates are represented by powertype patterns and the resources are represented by classes. Figure 4 shows these concepts in the top. The templates and resources represent common traceability issues which were detected from the traceability state of the art analysis as it is stated in [27].

This minimal core set supports the creation of traceability methodologies for any project features types, such as the application domain, project size, budget and particular requirements of the development process. The reason is that *TmM* is extended from *SEMDM* metamodel, which actually does not determine any development process model, methodology or life cycle. However, some other traceability items can be added to the core set thanks to the extension mechanisms provided by the definition of *TmM*, according to the project variations and specificities. TmM also includes the modeling of the interaction between the core items. Additionally, TmM includes the modeling for its process usage:

1. Project-specific traceability methodology definition
2. Project-specific traceability methodology enactment.

4.3 TmM core: templates and resources

The *TmM* templates, which are represented with powertype patterns in Fig. 4, are: traceability *work products* (marker *Twp*), *work units* (marker *Twu*), and *producers* (marker *Tp*). Each template category is subdivided in other patterns, to get the core set of traceability items from which will be defined the project-specific traceability methodologies. In Fig. 5, in terms of powertype patterns, work products are represented with the *TraceabilityWorkProduct* pattern, which is specialized into the *TraceabilityLink*, *LinkageRule*, and *TraceabilitySpecificationDocument* powertype patterns. The traceability producer is represented by the *TraceabilityProducer* powertype pattern, and the traceability role concept is represented by the *TraceabilityRole* powertype pattern. The *TraceabilityWorkUnit* pattern is a composition of traceability tasks, which are represented with the *TracingTask* powertype pattern. Note that the attributes for classes in the method level (top) are different, from the attributes in the project level (bottom).

In diagram of Fig. 5 again, classes in top are intended to support the methodology definition, and classes in bottom support the methodology usage. For instance, the *TraceabilityTypeKind* class has attributes to support the definition of traceability types before any link is created. Attributes such as the link description, the rule to create the link of such type, or the roles authorized to make changes to links of such type, are presented in this class. Whereas, the *TraceabilityType* class has attributes to manage the link during the software development, such as the identification of the source and target artifacts, indication whether the link is active, the version of the link or the dependency level between the linked artifacts (link weight).

The *TmM* resources, which are created during the methodology definition to support the traceability implementation during the project development, are represented as classes in Fig. 6, which are: *GranularityLevel*, *TraceabilityWeight*, *TraceabilityMetric* and *ArtifactTracingGuideline*.

TraceabilityLink powertype pattern represents all potential traceability link types to create links, and that might be defined in the traceability methodology. Types such as task, resource or goal dependency between two system artifacts, as well as, evolution, satisfaction or rationale types, to mention some. The traceability type definition will be according to the project characteristics, for instance trustworthy systems need to make emphasis on different variations of the dependency and rationale traceability types, to maintain a high level of system quality assurance.

Similarly, the *LinkageRule* pattern expresses all possible linkage rules that might be defined to automate the traceability links acquisition process. A linkage rule expresses a customized linkage rule, with a specific logics to create links of a given traceability type previously defined in the customized traceability methodology. The *TraceabilityRole* pattern expresses all the roles that control the traceability information access. Traceability data views that are built based on traces information, are presented to the stakeholders depending on their roles. The *TracingTask* pattern expresses the tasks to manage traceability links. A traceability task causes an action, in this case the *LinkUpdating*, to create, delete or update traces.

The *TraceabilitySpecificationDocument* pattern represent the traceability document, which specifies the traceability methodology information. This includes traceability resources according to project features such as granularity levels of the artifacts to link, weights to indicate the dependency level between two traced artifacts, traceability metrics, or the guidelines to indicate the right artifacts to link.

The instantiation process is detailed in the next section, and describes how to use the top and bottom classes through the three-layer hierarchy of TmM, to generate traceability methodologies.

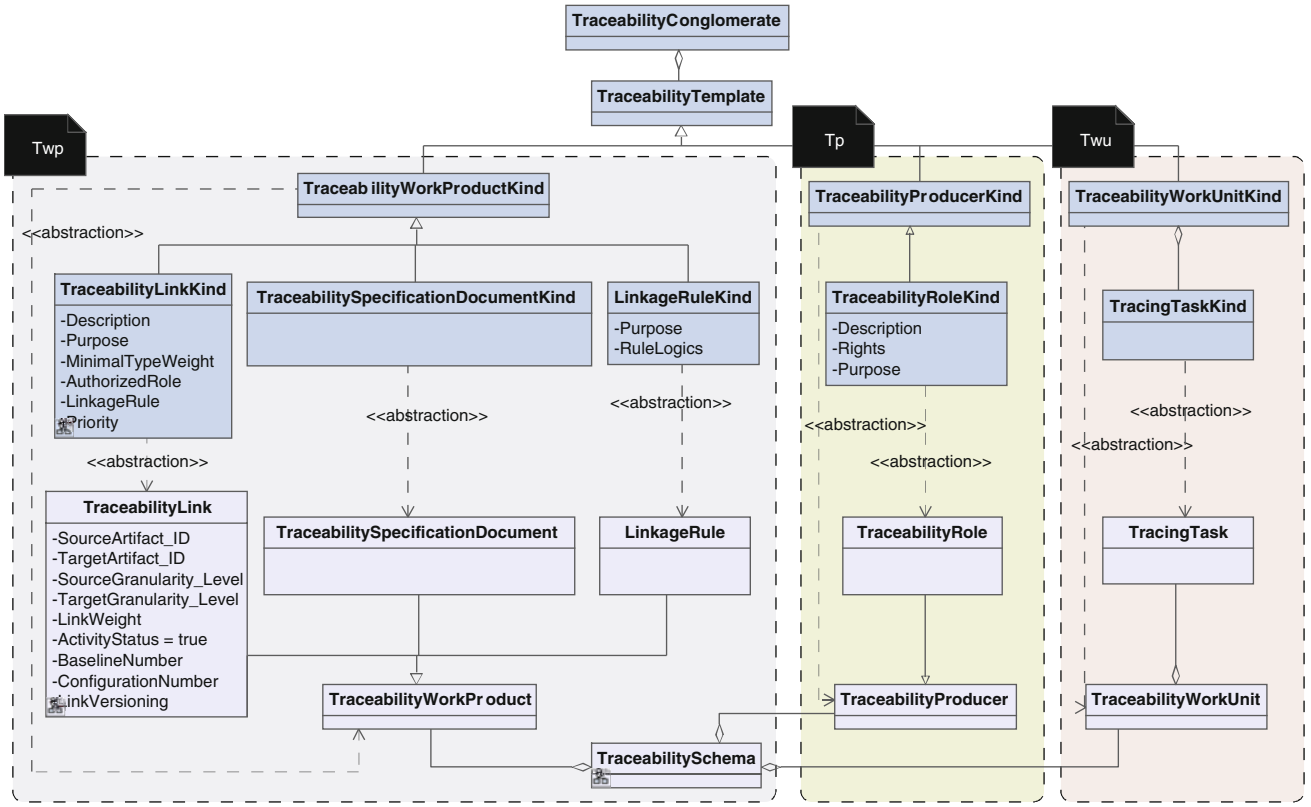

Fig. 5 TmM core (part I): traceability templates represented with powertype patterns

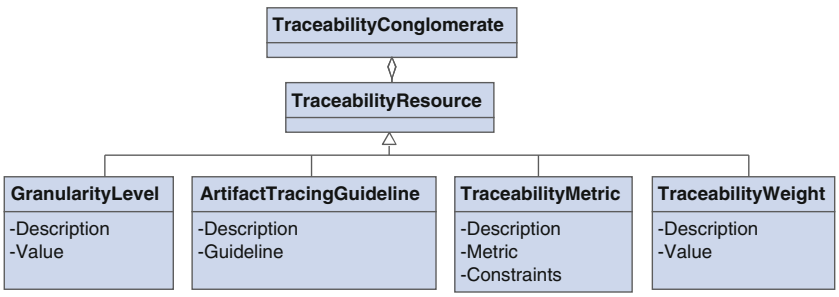

Fig. 6 TmM core (part II): traceability resources represented with classes

4.4 TmM usage process

This section briefly introduces the process followed to instantiate the TmM metamodel patterns, to define the traceability methodology items. Since TmM is a SEMDM extension, the process followed to instantiate the TmM patterns will be aligned with the SEMDM instantiation process.

Figure 7 structures the usage process for the TmM approach, in three-layer abstraction levels. In the metamodel layer the TmM metamodel is defined, by the method engineer. Here the project items for traceability are modelled using powertype patterns (marker A, Fig. 7). The powertype patterns are: *work products* identified with *Twp*, *work units* identified with *Twu*, and *producers* identified with *Tp*.

In the methodology layer, several stakeholders participate in the project features definition (marker B, Fig. 7), which is made according to the bussines requirements stated by the customer. The product features determine which traceability work products will be used to produce the final system. Hence, based on the project features, the method engineer defines the traceability items required to support the system development (process 1, Fig. 7). Additionally, the traceability metamodel indicates how the traceability items interact with the system items, during the software development effort. Here, the outcome of the *project-specific traceability methodology definition* process will be the specific items to implement traceability in the project (marker C, Fig. 7).

In the project layer, the developer uses the traceability methodology items previously defined (process 2, Fig. 7). Here, several traceability objects are produced (marker D, Fig. 7). For instance the links will be created, and data are used to produce views to present the traceability information

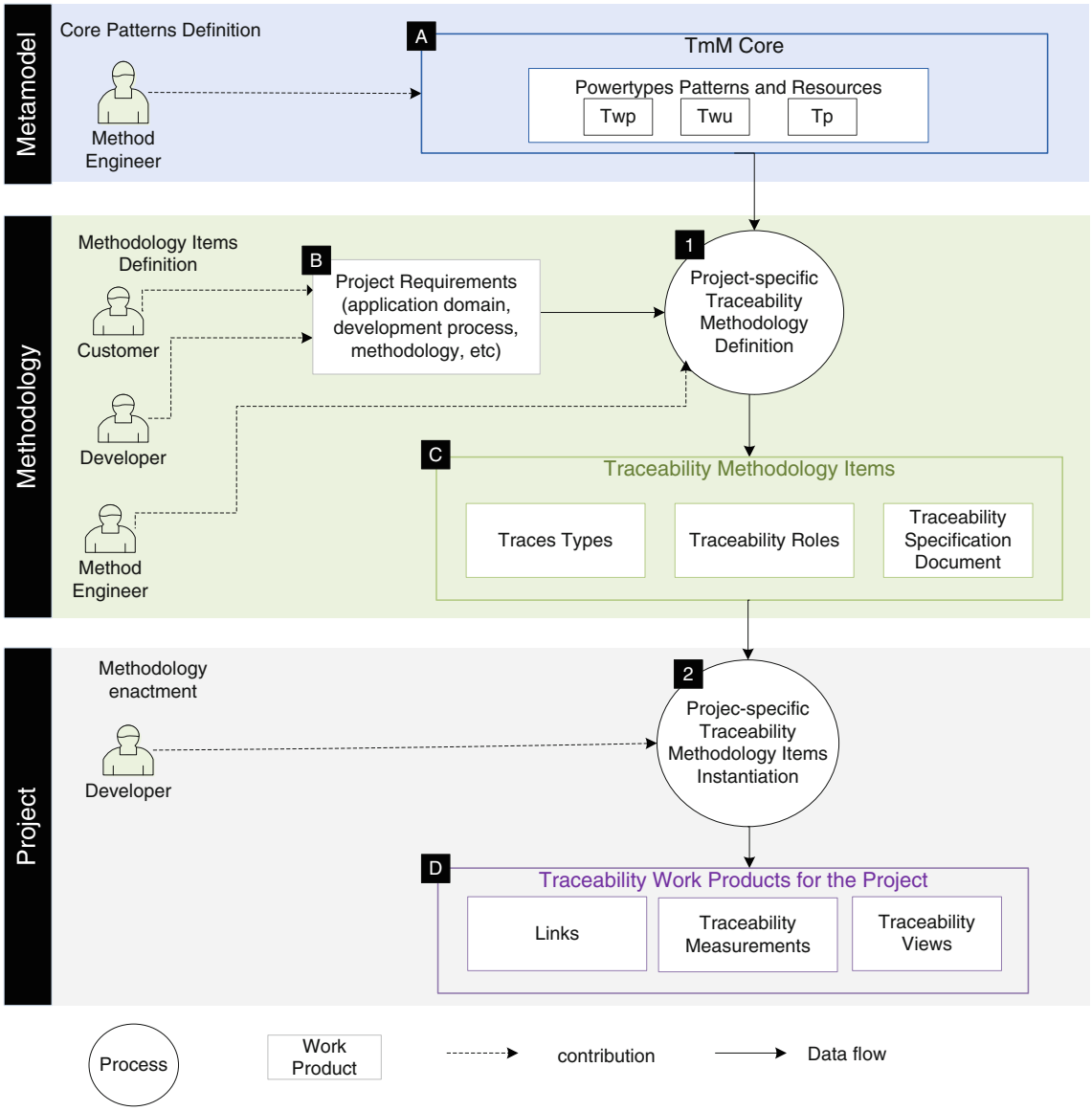

Fig. 7 TmM usage process

to the stakeholders, according to their roles defined in the project-specific traceability methodology. Moreover, some traceability measurements are implemented to collect metrics which are used to check for the links data consistency.

Note that, as the TmM metamodel is a set of power-type patterns and classes, then there is no constraint to any development software life cycle or methodology. Hence, TmM widely supports the creation of the specific traceability methodology, for an agile method. Even more, the TmM core set of patterns fulfills the requirements stated in Sect. 3 that were claimed as fundamental in any traceability model to effectively boost agility. The *TraceabilityLink* pattern supports to customize the traceability types for a project; the *LinkageRule* patterns is the platform which a priory restricts to define the logics or rule to create links for customized traceability types; and finally the *TraceabilityRole* patterns restricts, during the roles definition, the rights that the stakeholders can have, even before starting the project.

5 Case study

This case study discusses how some of the features provided by TmM can significantly improve the support provided by traceability models to agile methods, which are focused on testing. A strong testing practice is essential if one is getting on board on agile processes, this is the case of TDD and SDD which take advantage of several testing techniques to build systems. TDD and SDD use FIT, a frequently used language to define functional requirements as tests [52], or the use of business requirements as storytests [55]. This case study

particularly is centered on SDD and presents how TmM is used to systematically create a traceability methodology with those features described in Sect. 3.

5.1 TraceabilityLink instantiation

Storytests are an alternative to detailed requirements [55, p. 3]. The system functionality expressed with a storytest is allocated into code components, related by traces indicating that such storytest *tests* those components. This is sufficient to perform testing through the story tests. However, there is a lack of traces indicating degrees of satisfaction of the storytest understood as a requirement.

Storytests help developers drive the overall, iterative implementation of the new functionality. Developers take a storytest as the starting point for their next piece of work, and use TDD when they drive changes in the application code [55]. However, if impact analysis for a change in a storytest is performed, the lack of specific traces makes the impact estimation more difficult. There are no specific traces which indicate which components realize or satisfy a given storytest, seen as a requirement. As a result, impact estimation becomes an exhaustive manual task. Even more, one of the SDD challenges is to keep storytests consistent with the underlying application code's structure and naming [55, pg. 8]. Then satisfaction traces, and not only test traces, are strongly necessary, and a traceability type with agile features must be defined to adequately support SDD. The aim is to make the change impact estimation task faster and more accurate.

With this objective, the traceability type called *satisfies_tests* is defined using the *TraceabilityLink* pattern, to relate code components to storytest bidirectionally. Thus, the *satisfies_tests* type has the following semantics: on the one hand, the component-storytest path indicates that the underlying code component *satisfies* the test seen as requirement. On the other hand, the storytest-component path indicates that the storytest *tests* the underlying code component.

The objective of a traditional *test* type is different; a link would simply indicate *is_tested_by* or *tests*. Hence, using a relation of the *satisfies_tests* type it will be possible to relate a code item with a test item, having two kinds of relationships at the same time: *satisfies* and *is_tested_by*. A problem is evident when a storytest is related to the underlying components which realize it, using only the traces of the *test* type: the impact must be estimated following only the *test* traces. This restricts the use of satisfaction degrees to indicate to what extent a code component satisfies the given storytest. Then, estimations about the coverage for the storytests will be impossible to make.

Figure 8 shows the *satisfies_tests* types already instantiated from the *TraceabilityLink* pattern (marker A, metamodel layer). The *satisfies_tests* type is an item of the generated traceability methodology (marker 1, method layer). From the *satisfies_tests* type, traceability links are created to relate code components to storytests at the project layer (marker D, project layer).

Current traceability methodologies, as those indicated in Sects. 2.1 and 6, could not support effectively the traceability for SDD. None of them consider to link a test as it were a requirement, to the underlying code which realize it. All of them, foster the use of requirements properly agreed and stated in a Requirements Analysis stage.

5.2 TraceabilityRole pattern instantiation

In SDD, a FIT-style specification can be regarded as both (1) a requirement and (2) a test, and it is frequently executed, in principle, by programmers, but written by the on-site customers. However, the requirements and tests will probably evolve while the system is developed. If continuous integration and rigorous testing are practiced, FIT-style requirements should be consistent with the produced code, through traceability support. Otherwise, the build will fail [52]. In this dynamic context, the on-site customer who writes and checks storytests in FIT language must be capable of modifying traces between a requirement and the refined requirements items, allocated in user-stories. In this sense, on-site stakeholders are considered actual developers, who are performing uncommon tasks in traditional development.

Similarly, code is continuously modified by programmers who do not know how to modify the storytests, due to these one are high-level requirements created by customers. Then, a process goal must be necessary to keep consistency between FIT-style specifications and code. With this aim in mind, the *SDD_tester* role is defined as the person responsible for writing, modifying, and checking storytests, and for refining them into fine-grained story-tests. This role can also be granted with the rights to access and modify the traces between storytests, and then, producers with this role will be the only people responsible for modifying the links between story-tests.

Therefore, the *SDD_tester* role must be defined as part of the traceability methodology. The *SDD_tester* role definition enables to control the changes throughout the traceability implementation, by limiting the actions the customer can take over *only* the traces between storytests. Then, the security over the rest of the traceability items is increased, and hence, the system information consistency is improved.

Currently, the change control for system artifacts and links, is managed by a requirements management or a change management tool. However, those tools have pre-defined traceability types and roles and they do not offer facilities to create new roles or types. It seems obvious just to add rights to the customer role to modify specific storytests, but that it is unpractical and time consuming. Then, the *SDD_tester* role

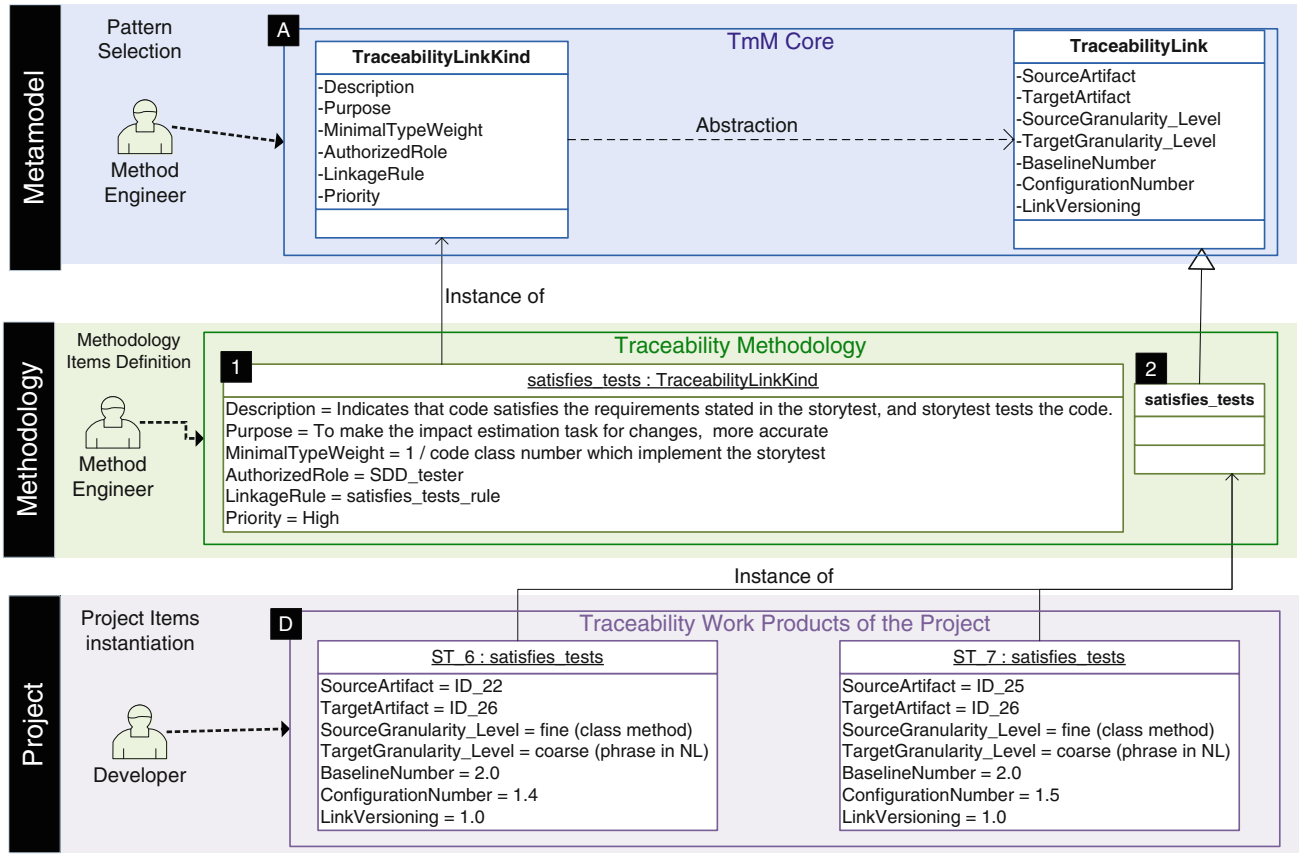

Fig. 8 Instantiating the the TraceabilityLink powertype pattern

is a required issue in a traceability methodology to adequately support SDD.

In terms of the TmM metamodel, subsection 4.3 introduces the *TraceabilityRole* pattern to define the applicable roles for the traceability activities in a project. Figure 9 shows how to use the *TraceabilityRole* pattern to define the *SDD_tester* role, as follows:

Figure 9 shows the *SDD_tester* role instantiated from the *TraceabilityRole* pattern (marker A, metamodel layer). The *SDD_tester* is an item of the generated traceability methodology (marker 1, method layer). Particular roles to manage different storytest types can be defined, such as the *sad-pad_Storytest_tester* role (marker D, project layer). This specific role is defined to edit traces between *sad-path workflow* storytests. This storytest type accounts for an action's failure, such as when a user violates business constraints. In [55, pg. 73] describes *sad-path workflow* storytests and other storytests types in detail.

5.3 LinkageRule pattern instantiation

Storytests have a meaning duality: requirements and tests; conversely, code items can include interfaces, components, classes or even processes. A linkage rule specifies conditional rules for artifacts relations in an automatic way, indicating the potential artifact types to be linked. Thus, the *satisfies_tests_Rule* has the logics to support the *satisfies_tests* links acquisition. The rule follows this logic: IF class, method, component realizes a storytest THEN link the item to the storytest USING the *satisfies_tests* type. It must be pointed out that a storytest could have *several* code items which realize it, and a code item could totally or partially realize a functionality required by a *unique* storytest.

Even though this rule might look like a simple conditional statement, once implemented in an automated tool or environment, it will reduce effort during the acquisition link task. This is because the rule works as a constraint for the traces between storytests and code using *only* the *satisfies_tests* type. Then, the rule prevents the possibility of relating storytests and code through the links of the *tests* type.

If *is_tested_by* links were exclusively used to relate a storytest to the code that realizes it, change impact effort estimation would be a more manual and exhaustive task and, probably, a less precise analysis. In this case no indication about the degree of satisfaction offered by the underlying code would be provided. Other useful information for effort estimation can not be obtained, such as the granularity levels of the artifacts linked and the dependency degree between

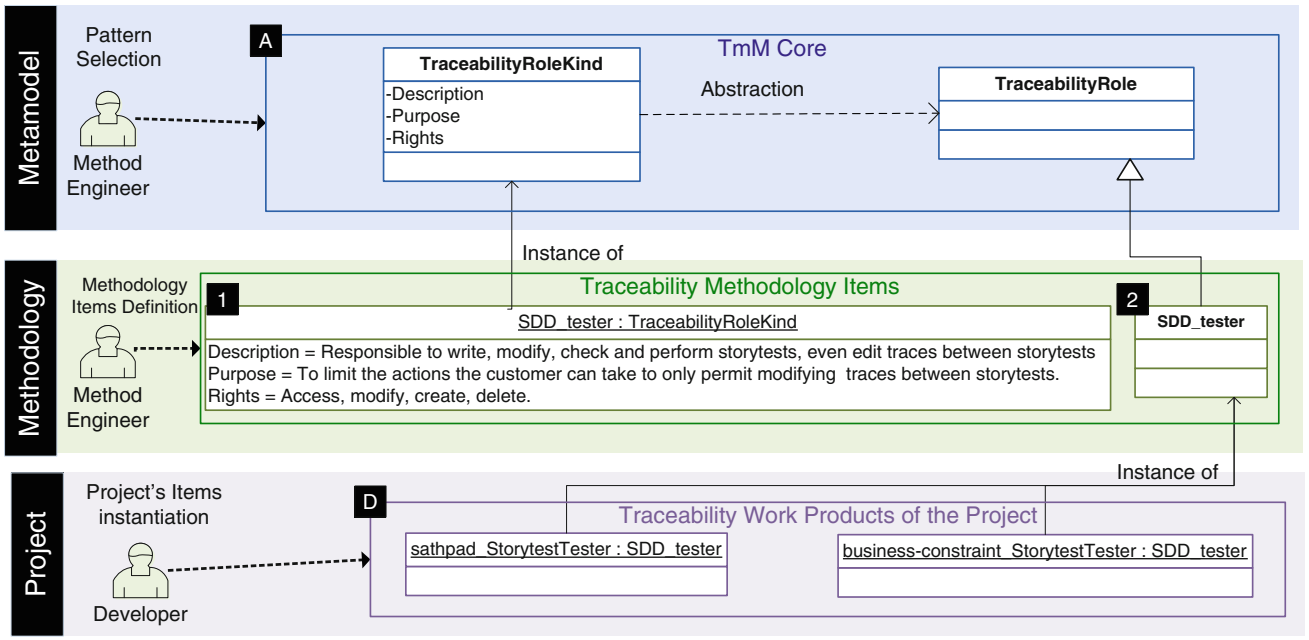

Fig. 9 Instantiating the TraceabilityRole powertype pattern

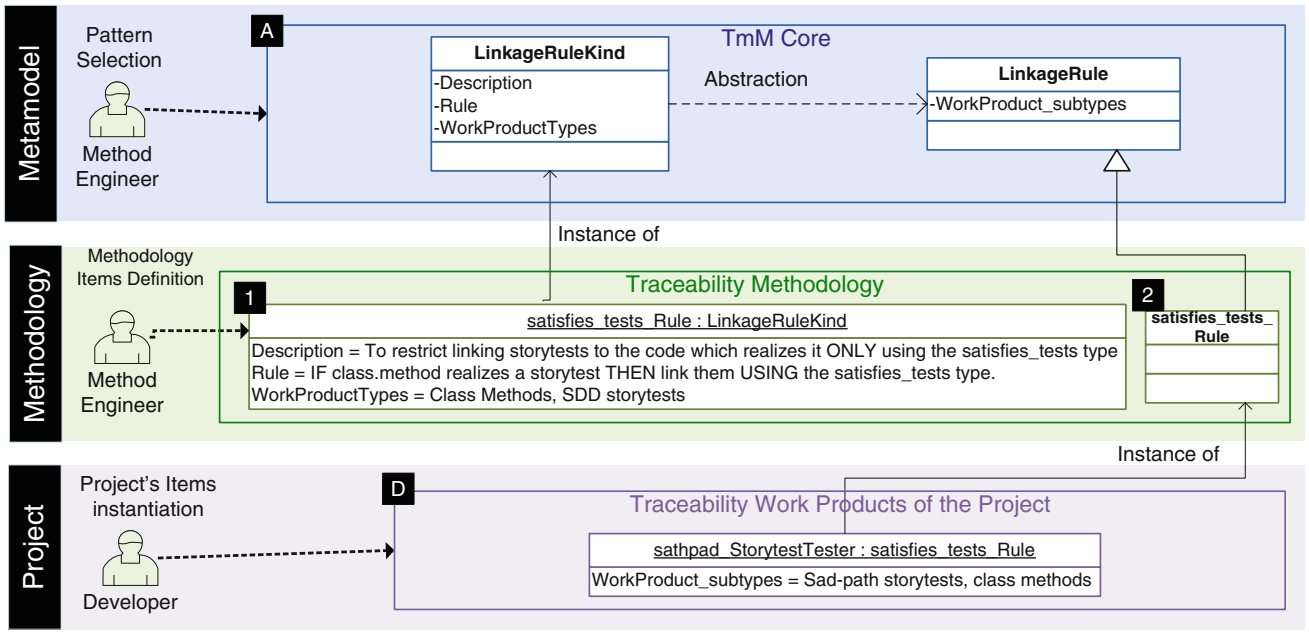

Fig. 10 Instantiating the LinkageRule powertype pattern

them. Then, a rule such as *satisfies_tests* will ensure a correct semantics for the relationship between storytests and code. In this sense, the linkage rules provided in TmM are, therefore, less constrained than in other approaches, such as the already discussed [13,15,25,24,66,70,71]. TmM widely makes it possible the creation of specific rules for an agile-featured project, as it is the case of SDD projects.

Figure 10 shows how to use the *LinkageRule* pattern, defined in Sect. 4.3 to specify linkage rules for a project (marker A, metamodel layer). The *LinkageRule* pattern is used to define the *satisfies_tests_rule*, to create traces between code components to storytest bidirectionally (marker 1, method layer). The *satisfies_tests_rule* is an item of the generated traceability methodology.

Particular linkage rules will be created from the *satisfies_tests_rule*, such as the *sad-pad_Storytest_to_code* rule (marker D, project layer). This sub-rule relates *sad-path workflow* storytests to the underlying code which realizes

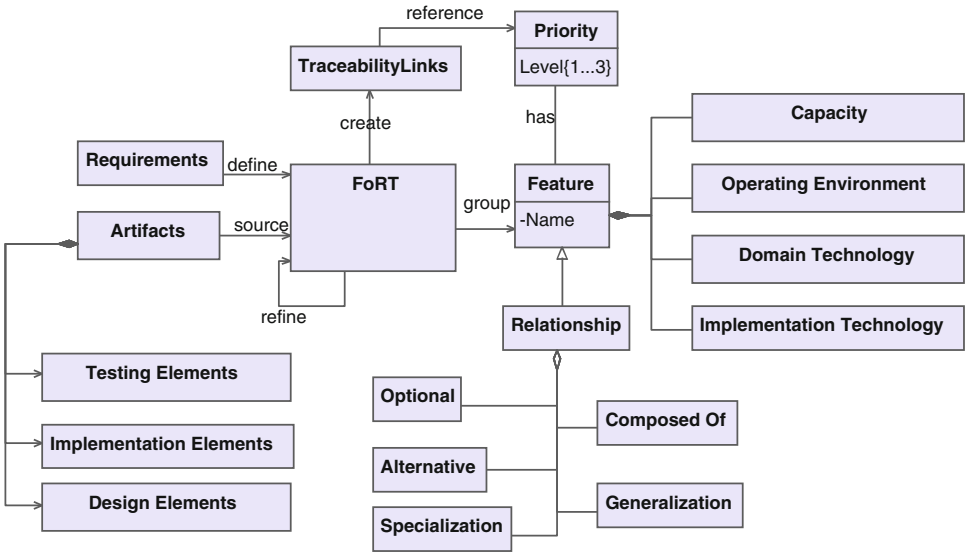

Fig. 11 A feature-oriented requirement tracing meta-model based on [2, Fig. 1] and [3, Fig. 1]

them. For more details of the *sad-path workflow* storytest, and other storytest types, see [55, pg. 73].

6 Related literature

Ahn and Chong in [2,3] introduce a meta-model for feature-oriented requirements tracing, shown in Fig. 11 based on [2, Fig. 1] and on [3, Fig. 3]. They perform feature oriented requirement tracing from user requirements to features, creating the correspondings traceability links by connecting artifacts to features. They also present an overview of a feature-oriented requirement tracing process. Some commonalities can be found with the approach adopted within this paper. Firstly, they use features with priorities, as an intermediate item between requirements and design artifacts, to support impact analysis. These features work as link attributes, similar to the link weight attribute of the *TraceabilityLink* pattern. Secondly, they use granularity levels for implementing artifacts. However, even though this is an interesting approach to support risk and value-based impact analysis, the proposal has a limitation: the metamodel is strongly influenced by a traditional development, in which requirements are artifacts and have nothing to do with tests. As described in both figures, the links are restricted to relate requirements to design, implementation and testing artifacts. It is not possible to relate a test to code, to indicate that such code realizes requirements specified as a test. Thus, the metamodel instantiation to provide traceability for agile methodologies, such as SDD or TDD, is not supported following this approach's metamodel.

Ramesh and Jarke [62] present traceability sub-models specific to development stages: the requirements management, rational, design allocation and compliance verification models, described in Figs. 3, 4, 5 and 6, respectively in that paper. The classification of these models presents a

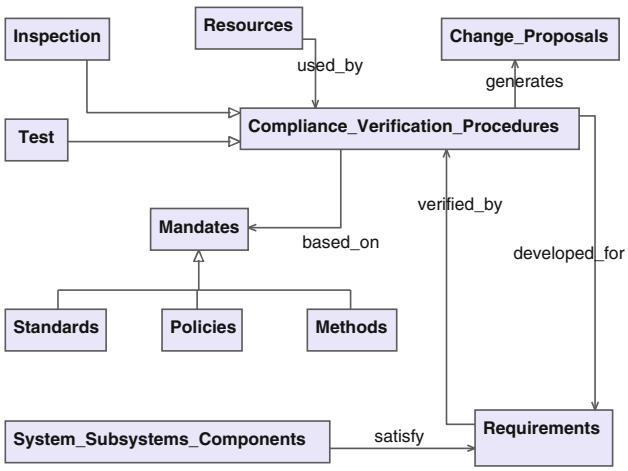

Fig. 12 Compliance verification submodel based on [62, Fig. 6]

clear dependency on traditional development stages; even more, the compliance verification model, shown in Fig. 12 based on [62, Fig. 6], clearly separates the *requirement* artifacts type from the other artifacts types, referring to them in the model as *system components*. In fact, the compliance verification sub-model describes explicitly the traceability link types as *satisfy* to indicate that system components satisfy a requirement, and *develop_for* to indicate that a test was designed, according to compliance verification procedures, to test a requirement. Thus, even though this is a model frequently referenced by many researches of the requirements engineering community, it does not support an agile methodology which specifies a requirement as a test.

In [71], the authors present a conceptual trace model depicted in Fig. 13 based on [71, Fig. 5], which describes traces acquisition. This approach gives detailed guidance on which traces should be established to support impact analysis at a later stage. The approach defines who should establish

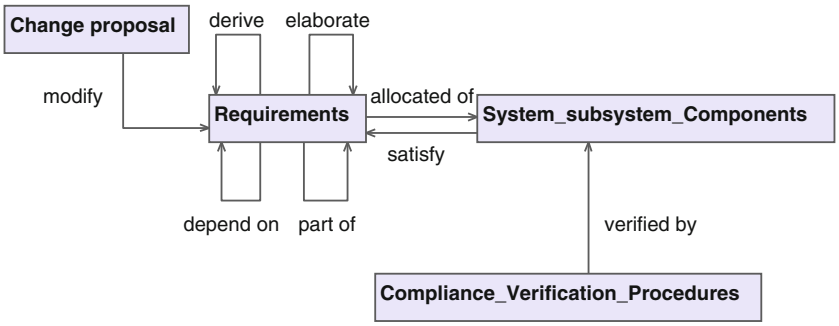

Fig. 13 Conceptual trace model based on [71, Fig. 5]

traces and when, along with who should analyze traces and how. An advantage of this approach is that it is supported by their authoring tool: QuaTrace, which is based on the authors' metamodel to perform traceability. The authors describe that their approach brings about traces to be established, analyzed, and maintained effectively and eficiently. However, the metamodel is constrained in the sense that it relates requirements to system components, and verification procedures explicitly to verify such components. Requirements are never considered to be tests at the same time, as is required in some agile processes. Again, this traceability model presents a strong dependency on a traditional development cycle, limiting its use in supporting agile methodologies.

All approaches analyzed fail to offer a platform to define customized traceability methodology items, such as special traceability link types or particular linkage rules to support the links acquisition of such special traceability types. The traceability types provided in those approaches are completely established and fixed, since the provided traceability models determine what kind of traceability types can be included in traceability implementation. They do not provide, therefore, traceability types general enough to create customized types. This is the gap that TmM is covering. Our approach is designed in three modeling levels, the metamodel, methodology and enactment levels, which provide the platform to design particular traceability types, roles and linkage rules along with traceability resources. The traceability patterns, which cover all the metamodeling levels, are a solution to develop customized traceability methodology items, without constraining our model to specific life cycle, process models, or application domain. Therefore, a traceability methodology with agile characteristics is totally supported following a TmM instantiation.

7 Conclusion and future work

This paper has studied how some concepts in traditional and agile processes may have different semantics. This paper justifies why traceability models should reflect the differences between traditional and agile processes; as it is discussed, otherwise, traceability models will be unable to effectively support agile processes. With this objective, a number of issues that traceability models should consider are identified: traceability links types, roles and linkage-rules. All these elements should be user-definable in traceability models. Predefined elements, as it is commonly found in literature, lead to complex models that do not really fit to real needs. To show how this could work in a model, an emerging metamodel for defining traceability methodologies, called TmM is used. This metamodel specifically supports user-definable traceability links types, roles and linkage-rules. Thanks to the user-definable properties the support that can be provided to agile processes stakeholders is significantly wider, as it has been proved in a case study presented.

The case study shows that user-definable elements and extensibility properties provided by TmM, become essential to support issues such as the lack of a formal requirements specification in TDD. To show how this works, TmM is used to define a specific traceability link, role and linkage rule. These are perfectly adapted to the used agile methods, TDD and SDD. As it is explained in each Case Study subsection, these specific traceability products could not be defined with other current proposals, due to the pre-definition of the traceability items in the third-party models.

Another advantage of the proposed approach is that it will be possible to achieve a higher degree of automation. In this sense it fosters a methodical trace acquisition and maintenance process.

In conclusion, the TmM metamodel enables the definition of a project-specific traceability methodology. This is possible because TmM clearly separates the metamodel, methodology and project expertise domains. This is useful for agile processes, but not exclusively.

Future work includes improved support for agile processes specifically for automation, together with further validation of the TmM model. The semantics of links and linkage rules is being further studied.

Acknowledgments This research work has been partially sponsored by the Spanish MITyC (FLEXI ITEA2 FIT-340005-2007-37), MICINN (INNOSEP TIN2009-13849, DSDM TIN2007-00889-E) and MEC (OVAL/PM TIN2006-14840). Special thanks to the Mexican National Council of Science and Technology (CONACyT) for supporting this research as part of the Doctoral Studies Financing Program.

References

1. Abrahamsson P, Warsta J, Siponen MT, Ronkainen J (2003) New directions on agile methods: a comparative analysis. In: Proceedings of the international conference on software engineering (ICSE). IEEE Computer Society, Washington, pp 244–254. ISBN:0-7695-1877-X
2. Ahn S, Chong K (2006) A feature-oriented requirements tracing method: a study of cost-benefit analysis. In: Proceedings of the international conference on hybrid information technology (ICHIT). IEEE Computer Society, Washington. ISBN 0-7695-2674-8. doi:10.1109/ICHIT.2006.17
3. Ahn S, Chong K (2007) Requirements change management on feature-oriented requirements tracing. In: Proceedings of the international conference on computational science and its applications (ICCSA), part II, pp 296–307
4. Aizenbud-Reshef N, Nolan BT, Rubin J, Shaham-Gafni Y (2006) Model traceability. IBM Syst J 45(3):515–526. ISSN:0018-8670
5. Alexander I, Robertson S, Maiden N (2005) What influences the requirements process in industry? A report on industrial practice. In: Proceedings of the 13th IEEE international requirements engineering conference (RE05). IEEE CS, pp 411–415
6. Antoniol G, Berenbach B, Egyed A, Ferguson S, Maletic J, Zisman A (2006) Problem statements and grand challenges in traceability. Technical report COET-GCT-06-01-0.9. Center of Excellence for Traceability, September 2006
7. Asuncion HU, François F, Taylor RN (2007) An end-to-end industrial software traceability tool. In: ESEC-FSE 2007. ACM, New York, pp 115–124. ISBN:978-1-59593-811-4
8. Beck (2002) Test driven development: by example. Addison-Wesley, Boston. ISBN:0321146530
9. Beck K, Andres C (2004) Extreme programming explained: embrace change, 2nd edn. Addison-Wesley. ISBN:0321278658
10. Berenbach B, Wolf T (2007) A unified requirements model; integrating features, use cases, requirements, requirements analysis and hazard analysis. In: Proceedings of the international conference on global software engineering (ICGSE). IEEE Computer Society, Washington, pp 197–203. ISBN:0-7695-2920-8
11. Brown AW (2004) Model driven architecture: principles and practice. Softw Syst Model 3(4):314–327
12. Cao L, Ramesh B (2008) Agile requirements engineering practices: an empirical study. IEEE Softw 25(1):60–67
13. Cerbah F, Euzenat J (2001) Using terminology extraction to improve traceability from formal models to textual requirements. Lect Notes Comput Sci 1959:115–126
14. Cleland-Huang J, Chang CK, Christensen M (2003) Event-based traceability for managing evolutionary change. IEEE Trans Softw Eng 29(9):796–810
15. Cleland-Huang J, Settimi R, Duan C, Zou X (2005) Utilizing supporting evidence to improve dynamic requirements traceability. In: Proceedings of the IEEE international conference on requirements engineering (RE). IEEE Computer Society, Washington, pp 135–144. ISBN:0-7695-2425-7
16. Cleland-Huang J, Berenbach B, Clark S, Settimi R, Romanova E (2007) Best practices for automated traceability. IEEE Comput 40(6):27–35. ISSN:0018-9162
17. Cockburn A (2000) Selecting a project's methodology. IEEE Softw 17(4):64–71. ISSN:0740-7459. doi:10.1109/52.854070
18. Cockburn AAR (1993) The impact of object-orientation on application development. IBM Syst J 32(3):420–444. ISSN:0018-8670
19. IEEE Computer Society Professional Practices Committee (2004) Guide to the software engineering body of knowledge (SWEBOK). IEEE
20. Cunningham W (2002) FIT: framework for integrated test. http://fit.c2.com
21. Dahlstedt Å, Persson A (2005) Requirements interdependencies: state of the art and future challenges, chapter 5. Springer, Berlin, pp 95–116. ISBN:10-3-540-25043-3, 13-978-3-540-25043-2
22. Dekhtyar A, Hayes JH, Larsen J (2007) Make the most of your time: how should the analyst work with automated traceability tools? In: Proceedings of the third international workshop on predictor models in software engineering (PROMISE). IEEE Computer Society, New Delhi, p 4. ISBN:0-7695-2954-2
23. Duan C, Cleland-Huang J (2007) Clustering support for automated tracing. In: Proceedings of the IEEE/ACM international conference on automated software engineering (ASE). ACM, New York, pp 244–253. ISBN:978-1-59593-882-4
24. Egyed A (2004) Resolving uncertainties during trace analysis. In: Proceedings of the ACM SIGSOFT international symposium on foundations of software engineering (SIGSOFT FSE). ACM Press, pp 3–12. ISBN:1-58113-855-5
25. Egyed A, Grunbacher P (2002) Automating requirements traceability: beyond the record and replay paradigm. In: Proceedings of the international conference on automated software engineering (ASE). IEEE, pp 163–171
26. Espinoza A (2009) An advanced traceability schema as a baseline to improve supporting life cycle processess. PhD thesis, Universidad Politecnica de Madrid. http://oa.upm.es/2557/
27. Espinoza A, Garbajosa J (2008) Tackling traceability challenges through modeling principles in methodologies underpinned by metamodels. In: Proceedings of the CEE-SET WiP. Oficyna Wydawnicza Politechniki Wroclawskiej, Brno, pp 41–54
28. Espinoza A, Garbajosa J (2008) A proposal for defining a set of basic items for project-specific traceability methodologies. In: Proceeding of 32nd annual IEEE software engineering workshop (SEW). IEEE Computer Society, Kassandra, pp 175–185. ISBN:978-0-7695-3617-0
29. Espinoza A, Alarcón PP, Garbajosa J (2006) Analyzing and systematizing current traceability schemas. In: O'Conner L (ed) Proceedings of the IEEE/NASA software engineering workshop (SEW). IEEE Computer Society, Columbia, pp 21–32. ISBN:0-7695-2624-1
30. Espinoza-Limon A, Garbajosa J (2005) The need for a unifying traceability scheme. In: Oldevik J, Aagedal J (eds) Proceedings: ECMDA traceability workshop (ECMDA-TW). SINTEF ICT, Nuremberg, pp 47–56. ISBN:82-14-03813-8
31. Evans MW (1989) The software factory. Wiley, New Jersey
32. Fitzgerald B, Hartnett G, Conboy K (2006) Customising agile methods to software practices at intel shannon. Eur J Inf Syst 15(2):200–213. ISSN:0960-085X. doi:10.1057/palgrave.ejis.3000605
33. Fletcher J, Cleland-Huang J (2006) Softgoal traceability patterns. In: Proceedings of the international symposium on software reliability engineering (ISSRE). IEEE Computer Society, Washington, pp 363–374. ISBN:0-7695-2684-5
34. Gonzalez-Perez C, Henderson-Sellers B (2006) A powertype-based metamodelling framework. Softw Syst Model 5(1):72–90
35. Gonzalez-Perez C, Henderson-Sellers B (2007) Modelling software development methodologies: a conceptual foundation. J Syst Softw 80(11):1778–1796
36. Gotel OCZ, Finkelstein CW (1994) An analysis of the requirements traceability problem. In: Proceedings of the international conference on requirements engineering (RE). Colorado Springs. IEEE Computer Society Press, pp 94–102
37. Grünbacher P, Hofer C (2002) Complementing XP with requirements negotiation. In: Proceedings of the international conference

on eXtreme programming and agile processes in software engineering (XP), Alghero, Sardinia, Italy, pp 105–108
38. Hayes JH, Dekhtyar A, Osborne J (2003) Improving requirements tracing via information retrieval. In: Proceedings of the IEEE international conference on requirements engineering (RE). IEEE Computer Society, Washington, p 138. ISBN:0-7695-1980-6
39. Henderson-Sellers B, Gonzalez-Perez C (2005) The rationale of powertype-based metamodelling to underpin software development methodologies. In: Proceedings: Asia-Pacific conference on conceptual modelling (APCCM'05). Australian Computer Society, Darlinghurst, pp 7–16. ISBN:1-920-68225-2
40. IEEE (1990) IEEE Std 610.12-1990 IEEE standard glossary of software engineering terminology. Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York
41. ISO/IEC (2008) ISO/IEC 12207:2008 systems and software engineering—software life cycle processes. ISO/IEC
42. ISO/IEC 24744 (2007) ISO/IEC 24744:2007 software engineering—metamodel for development methodologies. ISO/IEC
43. Jackson J (1991) A keyphrase based traceability scheme. In: IEE colloquium, computing and control division, professional group Cl., pp 2/1–2/4
44. Jane C-H, Habrat R (2007) Visual support in automated tracing. In: Proceedings of the second international workshop on requirements engineering visualization, 2007 (REV 2007). IEEE Computer Society, New Delhi, p 4. ISBN:978-0-7695-3248-6
45. Kaindl H (1993) The missing link in requirements engineering. ACM SIGSOFT Softw Eng Notes 18(2):30–39
46. Lago P, Muccini H, van Vliet H (2009) A scoped approach to traceability management. J Syst Softw 82(1):168–182. ISSN:0164-1212. doi:10.1016/j.jss.2008.08.026
47. Lefering M (1993) An incremental integration tool between requirements engineering and programming in the large. In: Proceedings of the IEEE international symposium on requirements engineering, 4–6 Jan 1993. IEEE, San Diego, pp 82–89
48. Lin J, Lin CC, Cleland-Huang J, Settimi R, Amaya J, Bedford G, Berenbach B, Ben Khadra O, Duan C, Zou X (2006) Poirot: a distributed tool supporting enterprise-wide automated traceability. In: Proceedings of the 14th IEEE international requirements engineering conference (RE'06). IEEE Computer Society, Washington. ISBN:0-7695-2555-5
49. De Lucia A, Fasano F, Oliveto R, Tortora G (2007) Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans Softw Eng Methodol 16(4):13. ISSN:1049-331X
50. Maeder P, Philippow I, Riebisch M (2007) A traceability link model for the unified process. In: Proceedings of the eighth ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing (SNPD 2007). IEEE Computer Society, Washington, pp 700–705. ISBN:0-7695-2909-7
51. Marcus A, Xie X, Poshyvanyk D (2005) When and how to visualize traceability links? In: TEFSE '05: Proceedings of the 3rd international workshop on traceability in emerging forms of software engineering. ACM, New York, pp 56–61. ISBN:1-59593-243-7. doi:10.1145/1107656.1107669
52. Martin RC, Melnik G (2008) Tests and requirements, requirements and tests: a möbius strip. IEEE Softw 25(1):54–59
53. Merisalo-Rantanen H, Tuunanen T, Rossi M (2005) Is extreme programming just old wine in new bottles: a comparison of two cases. J Database Manag 16(4):41–61
54. Morris SJ, Gotel OCZ (2007) Model or mould? A challenge for better traceability. In: Proceedings of the international workshop on modeling in software engineering (MISE). IEEE Computer Society, Washington, p 1. ISBN:0-7695-2953-4
55. Mugridge R (2008) Managing agile project requirements with storytest-driven development. IEEE Softw 25(1):68–75
56. Nawrocki JR, Jasiñski M, Walter B, Wojciechowski A (2002) Extreme programming modified: embrace requirements engineering practices. In: Proceedings: RE'02, pp 303–310
57. Pikkarainen M, Passoja U (2005) An approach for assessing suitability of agile solutions: a case study. In: Proceedings: XP 2005, pp 171–179
58. Pilgrim J, Vanhooff B, Schulz-Gerlach I, Berbers Y (2008) Constructing and visualizing transformation chains. In: ECMDA-FA '08: Proceedings of the 4th European conference on model driven architecture. Springer, Berlin, pp 17–32. ISBN:978-3-540-69095-5. doi:10.1007/978-3-540-69100-6_2
59. Pinheiro FAC (2003) Requirements traceability. In: Perspectives on software requirements. Springer, Berlin, pp 93–113
60. Pohl K (1996) PRO-ART: enabling requirements pre-traceability. In: Proceedings of the second international conference on requirements engineering. IEEE, pp 76–84
61. Pohl K, Dömges R, Jarke M (1997) Towards method-driven trace capture. In: Proceedings of the international conference on advanced information systems engineering (CAiSE '97). Springer, London, pp 103–116. ISBN:3-540-63107-0
62. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Softw Eng 27(1):58–93
63. Richardson J, Green J (2004) Automating traceability for generated software artifacts. In: Proceedings of the 19th IEEE international conference on automated software engineering (ASE '04). IEEE Computer Society, Washington, pp 24–33. ISBN:0-7695-2131-2
64. Van Schooenderwoert N, Morsicato R (2004) Taming the embedded tiger—agile test techniques for embedded software. In: Proceedings of the agile development conference (ADC'04). IEEE Computer Society, Washington, pp 120–126. ISBN:0-7695-2248-3
65. Schwaber K (2004) Agile project management with scrum. Microsoft Press, Redmond. ISBN:073561993X
66. Spanoudakis G (2002) Plausible and adaptive requirement traceability structures. In: Proceedings of the 14th international conference on software engineering and knowledge engineering (SEKE '02). ACM Press, New York, pp 135–142. ISBN:1-58113-556-4
67. Spanoudakis G, Zisman A, Pérez-Miñana E, Krause P (2004) Rule-based generation of requirements traceability relations. J Syst Softw 72(2):105–127
68. Tabares MS, Moreira A, Anaya R, Arango F, Araujo J (2007) A traceability method for crosscutting concerns with transformation rules. In: Proceedings of the 29th international conference on software engineering workshops (ICSEW '07). IEEE Computer Society, Washington. ISBN:0-7695-2830-9
69. Tekinerdogan B, Hofmann C, Aksit M (2007) Modeling traceability of concerns in architectural views. In: Proceedings of the 10th international workshop on aspect-oriented modeling (AOM '07). ACM, New York, pp 49–56. ISBN:978-1-59593-658-5
70. Volzer H, MacDonald A, Hanlon A, Lindsay P (2004) (SubCM): a tool for improved visibility of software change in an industrial setting. IEEE Trans Softw Eng 30(10):675–693. ISSN:0098-5589
71. von Knethen A, Grund M (2003) Quatrace: a tool environment for (semi-) automatic impact analysis based on traces. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, p 246. ISBN:0-7695-1905-9
72. Warden S, Shore J (2007) The art of agile development: with extreme programming. O'Reilly Media, Inc. ISBN:0596527675