Managing User Stories

Karin K. Breitman, Julio Cesar Sampaio do Prado Leite
karin, julio@inf.puc-rio.br
Departamento de Informática - PUC-Rio
Rua Marquês de São Vicente 225 - Gávea
Rio de Janeiro – RJ 22453-900
Brasil

Abstract:

Assuming that the system documentation produced by the Extreme Programming (XP) process is the actual code, we propose to enrich this documentation by aggregating the user stories (system requirements) to the final product, the code. In XP system requirements are elicited from clients using a template named user story cards that are further discarded. By simply annexing the user stories as they are to the correspondent piece of code they relate to, we would already have the following gains:

We go beyond annexing the user stories to code and propose a new organization to the information that is already elicited through the user stories. We base our proposal in our past experience developing software using scenarios. We are providing a scenario structure in which to capture and record the use stories that will provide

1. Introduction

It is a well know fact that agile development processes go by with doing little or no requirements engineering. In particular, if we take a look at the XP [Beck00] we observe that most of the requirements are gathered and represented in the format of user stories. There is very little information on what actually happens to the stories once they are turned into code. As requirements engineers we have a clear understanding that system requirements are persistent artifacts and that they should be kept available, even after system deployment, in order to guarantee traceability. Requirements also change, impacting directly the code they represent. Those changes ought to be made clear so that we can keep the code compatible and up to date with current requirements. XP does not address those issues directly, rather treats them informally. The justification lies in the prospective loss of time and effort that this task would consume. We argue that this need not to be, rather, most of this information can be kept and made available with the same effort and in time frame that XP developers have been devoting to the capture and analysis of user stories. Our idea is to enrich the XP documentation (code) by the aggregation of the system requirements (user stories). We propose a new organization to the user stories, that without demanding further effort, will naturally provide more process data and facilitate information retrieval.

Assuming that the system documentation produced by the Extreme Programming (XP) process is the actual code, we propose to enrich this documentation by aggregating the user stories (system requirements) to the final product, the code. In XP system requirements are elicited from clients using a template named user story cards that are further discarded. By simply annexing the user stories as they are to the correspondent piece of code they relate to, we would already have the following gains:

1. the storage of the information in a persistent and retrievable manner (the examples in [Beck00] are manuscript – hard to store and consult. (Since the user stories are the requirements, there should be no doubt that they ought to be persistent)
2. traceability – the stories capture information such as date, risk and priority. Interestingly enough, they do not capture ownership.
3. The possibility of deriving test cases from the user stories.

We go beyond annexing the user stories to code and propose a new organization to the information that is already elicited through the user stories. We base our proposal in our past experience developing software using scenarios. We are providing a scenario structure in which to capture and record the use stories that will provide

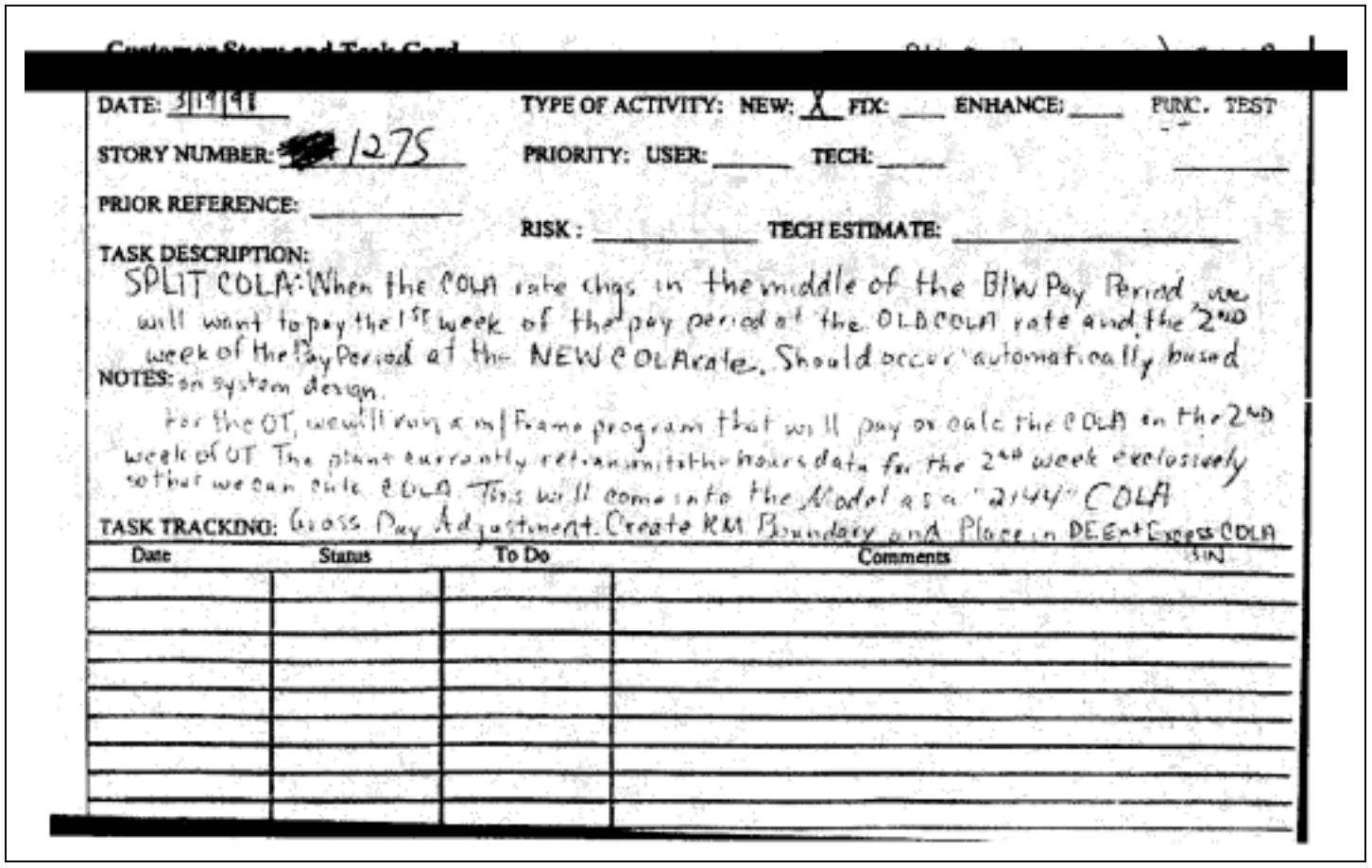
1. support to story evolution. It is clear that both the user stories and corresponding code suffer changes in the process. Those should be made explicit and easy to access.

2. basic configuration management capabilities, e.g., version control, tags.
3. Design rationale recording mechanisms. It is a fact that the most critical decisions are usually made during early development phases. Keeping record of the important ones is a sound practice.
4. More sophisticated traces, e.g., access to other stories with similar terminology, navigation through previous versions of the pair story/code, possible links to other artifacts, e.g., manuals, policies, questionnaires.
5. Allow for automatic consistency checking

central ideas of XP and detail the user stories. In the following section we introduce our proposal to the organization of the user stories in as scenarios. We discuss technological details and how our proposal can be implemented. We conclude by presenting our remarks and pointing to future research.

2. XP Overview

In a nutshell a XP team is comprised of 3 to 10 programmers and the client that should be on the development site to provide expertise[1]. The development room(s) are arranged in a way to promote the best


Customer Story and Task Card

DATE: 3/19/98 TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: [redacted]1275 PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___ RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:
SPLIT COLA: When the COLA rate chgs in the middle of the B/W Pay Period we will want to pay the 1ST week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.

NOTES: For the OT, we will run a mainframe program that will pay or calc the COLA on the 2ND week of OT. The plant currently retains in the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DE. Ent Excess COLA

Date	Status	To Do	Comments	SIN

Figure 1 – Customer story card

This organization borrows from our past experience using scenarios in software development. When compared, the two notations share some common ground, for the intention of both is describing a situation. By organizing a user story in the molds of a scenario we facilitate overall understanding and gain extra information on both context and functional plans. In the next section we introduce the

communication among the involved, among others the use of adjacent rooms, monitors arranged in a circle are common techniques. One very important concept is that there should be half the number of workstations as there

[1] Although the original proposal stresses the individual (one) client we believe it is proper to have more than one client present. One of us pointed out this aspect to be one of the most poignant problems in the XP approach [Leite01].

are programmers. The last is to promote team work, known as pair programming.

The development of the software is done in *iterations*. Each iteration is a three week period that results in running and tested code. The system is passed on to clients at the end of a release period, that usually comprises two to five iterations. That way the clients have an incoming stream of software every 6 to 15 weeks.

Directly quoting from Cockburn' scenario Agile Software Development "the unit of requirements gathering is the "user story"[2], user visible functionality that can be developed within one iteration. The client writes the stories for the iteration onto simple index cards. The client and programmers negotiate what will get done in the iteration the following way:

- The programmers estimate the time to complete each card
- The clients prioritize, alter, and de-scope as needed so that the most valuable stories are most likely to get done in the allotted time period [Cockburn02]."

In Figure 1 we show an example of a story card, as proposed in [Beck00].

Once the stories are selected, they are decomposed into programmable tasks and written in a white board, together with the time estimates for completion. Programmers implement the code in pairs and are constantly integrating the new code to the rest (a typical cycle would range from 15 minutes to a few hours). During the coding process the client(s) answers doubts, write tests to be run at the end of the iteration and work on a new round of stories for the next iteration. During the iterations the team hold daily meetings (standing, to keep short) where they discuss their working strategies. The result of the process is running code, delivered to the customer. In the next section we take a closer look into user stories.

3. User stories and requirements

User stories are the starting point in XP development. It is by choosing them that the customer kick starts the iteration process. They are defined as follow: "*one thing that the customer wants the system to do. Stories should be estimated at between one to five ideal programming weeks. Stories should be testable*."[Beck00]. Stories can be decomposed into tasks, quantifiable units of development effort. The decomposition is made by the programmers that also have to estimate how long it would take to implement each task. System development is a succession of such iterations where the requirements are continuously being defined by means of stories.

According to Beck, once the stories (and their task decomposition) are used, they are to be discarded. This corresponds to the *Embracing change and travel light* concept. In the author's idea there is no need to save the stories once they had their impact on the code because it (the code) is bound to change anyways. Another justification for throwing the stories away is the difficulty in keeping multiple forms of the same information synchronized.

We disagree with the author, first because the set of stories are the requirements documentation of the project. There is a plethora of arguments why the requirements documentation of a project is persistent, ranging from reuse to legal aspects, but the core argument in our discussion is maintaining traceability. On the synchronization aspect we disagree in two counts. First we do not think it difficult to keep the documents synchronized because if, for every major change in a story there will be a corresponding change in the code, it is enough to relate stories to code in a one on one relationship (old story-old code and new-story-new code). In fact, if we implemented a simple configuration management strategy we could maintain a history of changes and, at a very low cost, add more value to the process. For instance we will be able to measure the amount of changes requested per story, and find out if our customer is not changing his or her mind once too often!

In a second count we do not agree that the information forms are the same (customer story card versus code). If we look closely at Figure 1 we observe that the customer story contains information on priority, tracking (to other tasks), description (a natural language description of what it should *do*) and type. None of this information is going to be present in the code. In fact, the author is not clear even if the code will contain comments, explaining what it is supposed to do. Imagine a situation where a programmer is hired in the middle of the development process and he has to be briefed on the functionality of each piece of code because neither the story cards are available nor the code is commented (it would be as simple as importing the description from the story). The counterpart is also true, the code will add new information. Examples are technical decisions such as choice of database, programming language, variables, none of which is mentioned in the stories. Our point, once again, is that since we been through the effort of writing the stories and identifying the tasks, why not incorporate them to the final code? There are many

[2] Authors quote

advantages, traceability and future reuse are only some examples.

In our experience with requirements evolution we have observed that the incorporation of simple mechanisms can, sometimes, add enormous information value to the process at very little cost [Breitman98, Leite98, Breitman00]. Among others, there is **configuration management**, that allows keeping track of versions and the ability to visualize the history of persistent artifacts, creation of a **glossary of project terms**, guaranteeing that both developers and customers share a common understanding, aggregation of **information tags** to artifacts, that register information on ownership and authorization, and the addition of **rationale techniques**, that facilitate describing and understanding the decision making process. We believe that the present story card notation should be improved as to included such mechanisms. As such, we propose they adopt the scenario notation.

We present our idea in the format of a framework, understanding that the aggregation of such mechanisms to the story card representation incurs in very little additional efforts (from customers, who after all are the responsible for them).

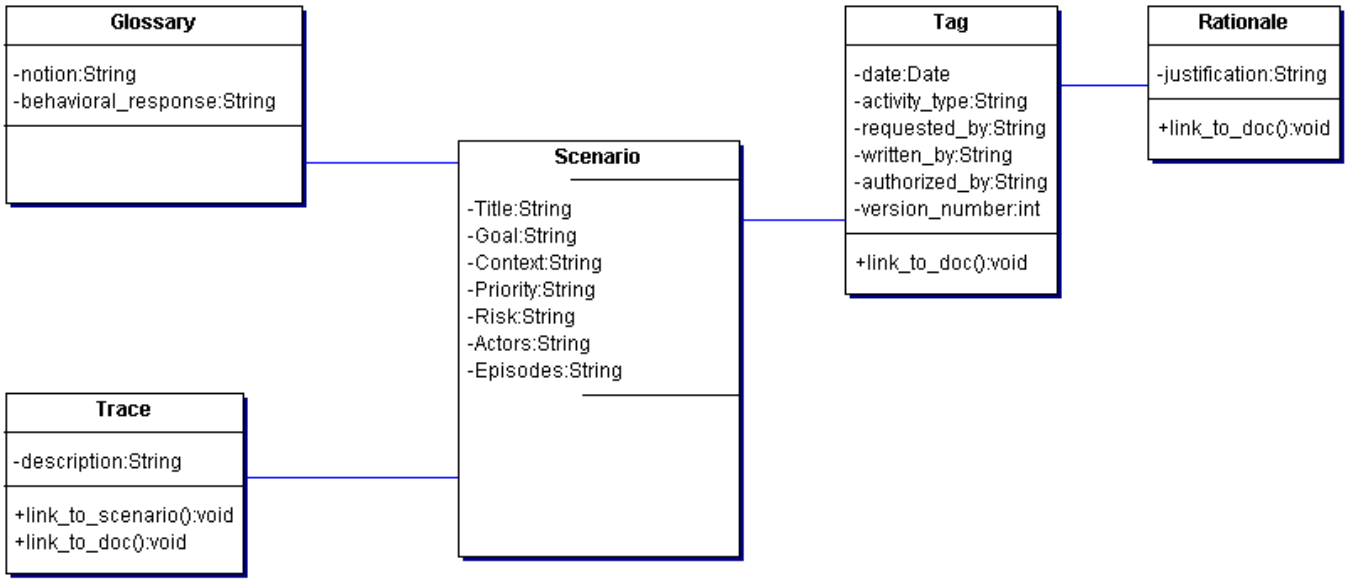


Figure 2 – Proposed scenario framework

The framework can be divided into two distinct parts. The core and the extra mechanisms offered. The core is represented by the class in the center, scenario, that contains all the attributes in the scenario notation proposed in [Leite98], namely title, goal, context, actors, resources and episodes, plus the risk and priority measurements, present in [Beck00]. The remaining classes represent the mechanisms that, once instantiated, will be responsible for capturing extra information on the process. They are, clockwise from the top left corner, glossary, tag, rationale and trace. It is important to note that some elements present in the customer story card representation, such as number and prior reference are implicitly taken care of in the scenario framework by the pair scenario_title + version and context, respectively. Also note that the component type of activity, also described in [Beck00] is in the tag class. Finally the task is represented by one or more episodes of the scenario.

The glossary class is responsible to map the vocabulary used by the customers (business) to commonly shared notions. Based on the language extended lexicon [Leite99] the glossary uses two definitions for each term, the denotation, notion of the symbol and the connotation (behavioral aspect) of the last that gives meaning to the symbol in relation to the context of use. The driving concepts of the glossary are closure and minimal vocabulary, i.e., symbols should be described using a minimal set of most used words and other symbols in the glossary can and should be used to describe new symbols. The glossary aims at keeping the consistency of the names used in the story cards and in the code. Take for instance the example depicted in Figure 1. What are the meanings of the terms Split Cola and OT? One may argue that the people involved in the project have incorporated those terms into their vocabularies, but what will happen if one programmer had to be substituted (or went on vacation) during development? What would be the effort associated to debriefing he or she on the project terminology? What if we assume that the story cards are

no longer available and the terms had made their way to the code?

The tag class essentially takes care of capturing the configuration management information associated to the scenario version. In this light we have the version number, date, type of activity and ownership. The ownership issue is of particular interest to system validation and, in the future, to maintain traceability to system requirements. We understand that most projects will have one or more clients in the development site. In our understanding the clients act as representatives of the different opinions (viewpoints) that people in his or her organization have. The in site-client is thus encumbered to describe and make clear the wishes of all groups in relation to the software. He or she of course knows who is responsible for each request. There is no reason why this information should be lost, the on-site client should be able to make it clear in the customer stories which individual or group is behind each request. This is why the class makes a distinction between the person/group who requested the functionality and the one who is writing the story card. This information will probably facilitate the validation process(es) and help pre-traceability, as defined by Gotel [Gotel97].

The rationale class, associated to the scenario tag, opens a venue to store design decisions. It may be kept in a free format style, natural language, or consist of a link to a document with a more structured format. These documents could be the output of any of the many rationale capturing techniques proposed in literature [Conklin88, Potts88, Carroll94, Gotel95, Potts96]. It seems clear that, in the XP context, story cards provide a means to improve communication and represent customer requests in a understandable way for both development and business (customers). They do not, however capture design decisions and the responsibilities that are crucial to maintaining system traceability. We understand that the adoption of any rationale capture technique is effort consuming and may represent a slow down in the development process. Our stand is that this is a decision to be taken by the client, who has to be offered the possibility of using a rationale capture technique and decide on the cost benefit of adopting the same.

Finally, the trace class presents the possibility of creating (manual) links to other customer story cards. One may want to relate to other stories for many reasons. One of them being the constraint that a story card should be feasible in the 1-5 ideal week time frame, so it is reasonable to suppose that decomposition will be needed occasionally. In this case, keeping track of related story card would be advisable. Another possibility is a situation in which a story card was deferred in favor of others with higher priorities. Perhaps it would be interesting to keep track of these stories in case there is time left during an iteration and their implementation might be reconsidered. The linking strategy is very simple and will appear as a list of “related stories” in the bottom of the scenario.

In this section we proposed a framework that takes care of those issues by allowing users to instantiate mechanisms to capture additional information. In the next section we show that the implementation of our ideas can be accomplished very simply using XML.

4. Implementation

The implementation of our proposal is very straightforward. We have created a DTD describing the components in the framework. The DTD is shown in Figure 3. It can be instantiated to an XML document that contains the elements of the framework. This serves as a reminder to users of what information is useful to the process. He or she will decide whether or not to take down some information and the granularity of its detail, based on specific project need, i.e., it may be the case that there is no rationale associated to the creation of a scenario version, so this tag may be left blank in the XML document.
The linking to the code could be done using a hypertext link, and the result could be viewed in any commercial browser. In the event the code is being generated in Java, there is also the possibility of inserting javadoc links in the code pointing to the related scenarios. In Figure 4 we implemented the example shown in Figure 1 using a possible instance of the DTD. In this example we included the information in Figure 1 plus some fictional data illustrating the instance of the classes.

Note that we included two traces, one to another story card (we call trace to similar objects homogeneous and to different one heterogeneous) and a trace to an document containing the values to be used in a calculation foreseen by the story card. There is also information on the ownership of the story. In this case the story card was requested by user Karin, written by user Otavio and authorized by a third user, Julio. This is not hard to imagine, as one would suppose that a change in calculations would have to be authorized by someone in management, and that this person is not likely to be the one assigned to be the on-site liaison to the programmers.
Finally there is the rationale to the change, that is explained by some new company policy, that was attached to the story card.

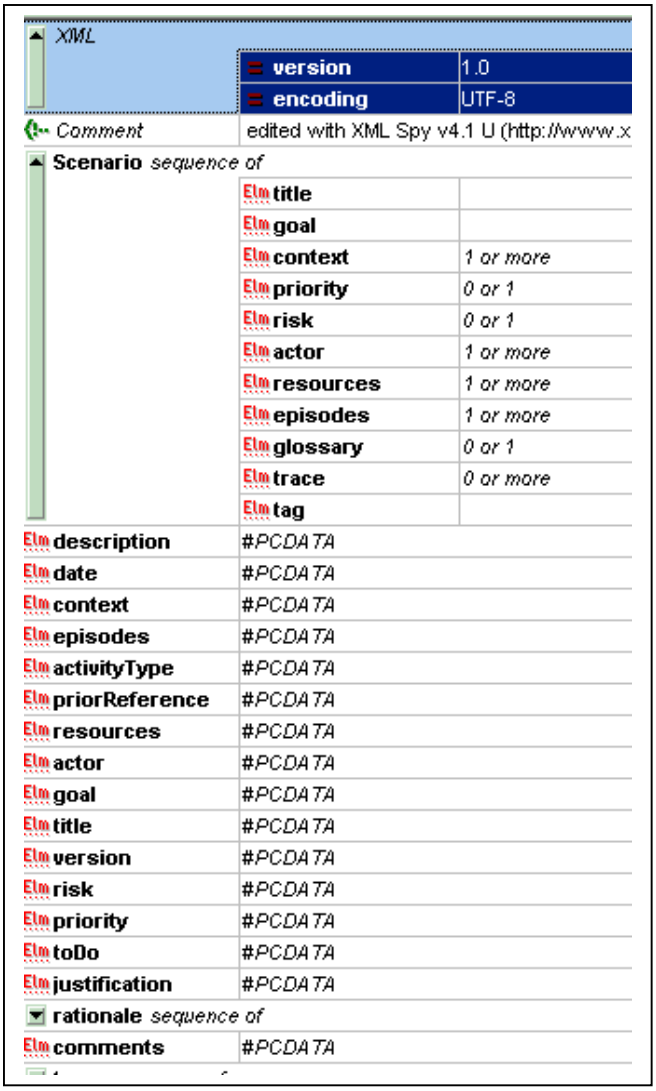


Figure 3 – DTD for the proposed framework

The integration to the code is also simple, it is enough to generate an XML document with the customer story card as header followed by the code. The outputs are very bare, e.g., as one can conclude from Figure 4. Our future plans include the creation of a scenario manager program to generate the HTML forms, thus providing a more user friendly interface to writing the scenarios.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE Scenario>
- <Scenario number="1275">
    <title>Calculate COLA rate</title>
      <goal>provide the COLA
    values</goal>
    <context>Blw pay period</context>
    <priority type=>high</priority>
    <risk>-----</risk>
    <actor> main frame program, plant,
      </actor>
    <resources>OLD COLA rate, NEW COLA
      rate</resources>
    <episodes>
        1. when the COLA rate changes in the
           middle of the Blw Pay Period pay the
           1st week of the pay period at the OLD
           COLA rate.
        2. pay 2nd. week of the Pay Period at the
           NEW.
        3. run a main frame program that will pay
           or calc the COLA on the 2nd week of
           OT.
        4. plant retransmits the hours data for the
           2nd week exclusively
        5. calculate COLA as a "2144"COLA gross
           pay adjustment.
        6. create RM Boundary.
        7. place in DEEntExcessCOLA BIN.
  </episodes>
- <trace type="homogeneous">
    <description>table containing the
      COLA rates</description>
    <link>
      http://www.xr.com/colarat
      es.xml</link>
  </trace>
-     <tag requestedBy="Karin"
    writtenBy="Otavio"
    authorizedBy="Julio">
      <date day="19" month="3"
      year="1998">03/19/1998</
      date>
                <activityType
      type="new">new</activityTyp
      e>
    <version>02</version>
  - <rationale>
      <justification>This procedure
        is the result of a new
        company policy regarding
        the calculations of the
        COLA rates.
        </justification>

        <link>http://www.abc.
        com/memorandum67.x
        ml</link>
    </rationale>
  </tag>

</Scenario>
```

Figure 4 – XML document illustrating the example in Figure 1

The XML implementation makes it possible to gain leverage by using emerging internet technologies. One possibility is to perform consistency checking among

scenarios. The Xlinkit site offers a consistency checking service among heterogeneous, distributed XML documents[xlinkit02]. We must remember that the XP process works in a succession of cycles, at the beginning of which a series of user stories, or scenarios as we propose, are written. The time frame between each cycle is 3 weeks and, it is only reasonable to expect that by the time new scenarios are written the previous ones are partially forgotten and inconsistencies may arise. Checking new scenarios against the set of previous generated ones (we include the ones that have been implemented and those that were deferred) may economize a lot of time and effort.

```
<consistencyrule>
    <forall                          var="x"
in="/Scenarios_Apr18">
        <forall                      var="y"
in="/Scenarios_May02”>
            <implies>
               <equal
op1="$x/Actor/text()"
op2="$y/Actor/text()"/>
            </implies>
        </forall>
    </forall>
</consistencyrule>
```

Figure 5 – Example of a rule

The Xlinkit engine operation is quite simple. Given a set of XML scenario documents and a set of consistency rules that should be applicable to the scenarios, the engine returns a set of Xlinks, i.e., links to the scenarios that do not satisfy the rules. The set of rules is written in a first order logic based language. The scenario representation naturally poses some constraints, e.g., an actor may appear in more than one scenario. To ensure consistency, the actors must always appear under the same name in the scenarios that he or she (or it) takes part. This rule seems straightforward enough but, there are complications if one considers that once scenarios are written in natural language, gender variations (specially in languages other than English) and plurals may come in the way. We exemplify this rule in Figure 5. Of course, that the list of rules will also be influenced by the application itself, i.e., the application domain will dictate the rules and constraints that must be observed. Writing the rules is quite straightforward and the Xlinkit package provides the rule language for writing your own constraints between documents and several examples. This is still a very naïve use of the potential of the consistency checking engine. In the future we intend to write rules to check consistency between the elements in the scenarios and their equivalent pieces of code. This requires the writing of more complex constraints to allow the comparison between the different data formats (natural language scenarios and programming language code).

5. Conclusion

In this paper we propose a framework in which to incorporate scenarios to code, considered the “document” in XP philosophy. We believe it to be fundamental to keep the scenarios once the system is running because they are the actual system requirements. There is no doubt that, in order to allow for traceability, the requirements documentation must be persistent and available at all times. The propose framework offers some additional information capture mechanisms, that at low cost, aggregate important information value to the process. The framework is user configurable, i.e., the user decides if it is cost effective to incorporate any of the offered mechanisms according to personal and project needs.

The implementation of our proposal uses XMI technology and it is quite straightforward. We believe its simplicity to be its major advantage. Future work include the development of a friendlier infra structure for instantiating and using the framework. Although not lacking in functionality, the instantiation and visualization tasks would greatly benefit from the implementation of a scenario authoring tool.

Another issue worth looking into is how to organize the code pieces once the system is ready. It is obvious that using 3-5 weeks development periods a several pieces are generated that have to be integrated in order to provide a consistent final product. The literature on use cases and scenarios provide some pointers.

Cockburn suggests that use cases should be structured and kept in a case planning table. The authors and goals are to be placed in the left most column an in the next columns one should record, if fit: business value, complexity, release, team, completeness, performance requirement and external interfaces [Cockburn02]. In the event that only part of the functionality is delivered, it should be noted in the planning table the first release in which the use case shows up and the final release in which the use case will be delivered in its entirety.

Kulak and Guiney baseline the approved use cases and the use case standards document, i.e., a document that contains an explanation of the contents of each use case. Among others this document includes information on the pre and post conditions, assumptions, exceptional and alternative paths of action, trigger, author and date. Their final use case output, exemplified throughout [Kulak00] contains all the above information.

In order to organize use cases, Armour defines dependency streams to model how individual use cases depend on each other based on pre and post conditions. The concept of a description was also added, in order to detail the information represented in the use case model. There are three levels of descriptions, initial, base and elaborated. They are used progressively to describe "the activities performed, alternative flows, conditional logic to document exceptions and alternative processing" [Armour01]. User cases are further organized in groups in what the author calls a business function package, that summarizes a responsibility of the system viewed from a functional perspective.

Leite proposes the creation of integration scenarios [Leite00]. These scenarios are artificially created to serve as a medium in which to relate scenarios using higher abstraction ones. According to the author the scenario construction process is middle out. We have reason to believe that this situation is quite common in an XP environment, and we foresee the need for integration user stories/scenarios, i.e., artificially generated artifacts with the sole purpose of combining a set of user stories/scenarios in one chunk of meaningful action.

Note that in all approaches, similarly to our proposal, extra information, in addition to the actual documents (use cases or scenarios) , is stored in order to organize the information. The underlying idea is the same, i.e., assure system traceability and allow for minimal domain oriented configuration management. We tackle those issues in a more structured way, allowing the users to define both their traceability and configuration management strategies by instantiating the scenario framework.

6. References

[Armour01] Armour, F.; Miller. - Advanced Use Case Modeling - Addison-Wesley, January 2001

[Beck00] – Beck, K. Extreme Programming Explained: Embrace Change – Addison Wesley – 2000.

[Breitman98] Breitman, K.; Leite, J.C.S.P. - Scenario Evolution: observations from a case study - Proceedings of the International Conference on Requirements Engineering, IEEE Computer Society Press, pp. 214-221, 1998.

[Breitman00] – Breitman, K.K.; Leite, J.C.S.P. **–** Scenario Evolution: A Closer View on Relationships – in Proceedings of the Fourth International Conference on Requirements Engineering (ICRE'00) – 2000.

[Carroll94] - Carroll, J.; Alpert, S.; Karat, J.; Van Deusen, M.; Rosson, M. – Raison nc. – 2000.
d'etre: capturing design history and *rationale* in multimedia narratives. Proceedings of Human Factors in Computing Systems (CHI94) – ACM Press - Boston, USA, 1994. pp. 192-197

[Cockburn02] – Cockburn, A. –Agile Software Development – The Agile Software Development Series – Addison Wesley – 2002.

[Conklin88] - Conklin, J. Begemann, M. – gIBIS: a hypertext tool for exploratory policy discussion. ACMTOOIS– 1988. pp. 303-331

[Gotel95] - Gotel, O. and Finkelstein, A. – Contribution Structures – in the *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95)* – York, March 27 to 29 – IEEE Computer Society Press, 1995, pp. 100-107.

[Gotel97] – Gotel, O. and Finkelstein, A. – Extended Requirements Traceability: Results from an Industrial Case Study– in the Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE'97) – Annapolis, January 6-10 – IEEE Computer Society Press - 1997, pp. 169-179.

[Kulak00] – Kulak, D.; Guiney, E. - Use Cases : Requirements in Context - Addison-Wesley, Paperback, Published May 2000.

[Leite00] - Leite, J.C.S.P., ., Hadad, G., Doorn, J., Kaplan, G. – scenario construction process - Requirements Engineering Journal vol(5) N.1 pp. 38-61 – Springer Verlag - 2000.

[Leite01]- Leite, J.C.S.P - Extreme Requirements

[Leite98] – Leite, J.C.S.P. et al. – Enhancing a Requirements Baseline with Scenarios - – Requirements Engineering Journal vol(2) pp. 184-198 – Springer Verlag - December, 1998.

[Leite99] - Leite, J.C.S.P.; Anchoring the requirements process on vocabulary, requirements capture, documentation and validation – Daghstuhl Seminar report 241- pp.13-14 – (1999) www.dagstuhl.de/DATA/Reports/99241.

[Potts88] - Potts, C. and Bruns, G. – Recording Reasons for Design Decisions – in Proceedings of the 10th International Conference on Software Engineering – IEEE Computer Society Press – April, 1988, pp.418-426.

[Potts96] - Potts, C. – Supporting Software Desing: Integration Design Methods and Design Rationale – in Design Rationale: Concepts, Techniques and Use, edited by T. Moran and J. Carroll, Lawrence Erlbaum Associates, Publishers – 1996. pp.295-321

[xlinkit02] – internet site – www.xlinkit.com.