Writing Coherent User Stories with Tool Support

Conference Paper *in* Lecture Notes in Computer Science · June 2005

DOI: 10.1007/11499053_38 · Source: DBLP

CITATIONS
6

READS
10

4 authors, including:

Michał Śmiałek
Warsaw University of Tech…
65 PUBLICATIONS **205** CITATIONS
SEE PROFILE

Tomasz Straszak
Warsaw University of Tech…
17 PUBLICATIONS **85** CITATIONS
SEE PROFILE

From User Stories to Code in One Day?

Michał Śmiałek

Warsaw University of Technology and Infovide S.A., Warsaw, Poland
smialek@iem.pw.edu.pl

Abstract. User stories in software engineering serve the purpose of discovering requirements and are used as units of system development. When applying stories in a project, two elements seem to be crucial: the ability to write coherent sequences of events and the ability to transform these sequences into code quickly and resourcefully. In this paper, these qualities are reflected in a notation that can be described as "stories with notions". This notation separates the story's sequence of events from the description of terms used in this sequence. Such a formal separation does not limit and rather enhances invention, at the same time rising the level of consistence, and facilitating translation into models of code. This translation maps domain notions into static code constructs (classes, interfaces) and also maps stories into dynamic sequences of messages. With such a mapping, programming becomes equivalent to skilled transformation of user stories, thus giving shorter development cycles.

1 Introduction

In Poland there is well known a novel written by Eliza Orzeszkowa, called "On the Niemen River". It is full of beautiful descriptions of the Polish-Lithuanian countryside around the Niemen river in the XIXth century. Of course, as for every normal novel, it also tells some story. The story is very suggestive and coherent, as all the characters move around this well-described piece of countryside, with rivers, lakes, roads, villages and mansions. The majority of Polish students remember this novel because of these rich descriptions of the nature.

At this moment one might ask what is the relation of Orzeszkowa's novel to software development. We shall try to argue that the relation is significant, especially in the area of requirements specification. Can we imagine a novel which has just the story and no description of the environment (people, places, landscape, and so on)? And when we quickly move to software engineering; can we imagine requirements specification that has just the story? An obvious answer is – no! Of course, when specifying the requirements for a software system we need to tell the developers a story which can be defined as an "account of incidents or events"[1] happening between a user (a role) and the system. However, this story has to be supported by descriptions of all the notions used therein.

Unfortunately, a majority of story writers in software engineering tend to mix stories with the descriptions of notions handled by their systems. These

[1] Merriam-Webster On-line Dictionary

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 38–47, 2005.

notion definitions are buried somewhere inside the stories. What is worse – the same notions are often described inconsistently in different stories (see [1]). This inconsistency is more or less acceptable when we write a novel, while it is totally disastrous when specifying a software system.

The requirements specification in software engineering has a clear purpose: to produce code that satisfies the **real** needs of the client. Supposing that we can discover these needs through stories with separated notions, we are still left with an important question: how to make code out of them? It seems obvious that we somehow need to transform stories into sequences of instructions in code, but what about the "descriptions of the nature"? Can we define some process of transforming stories and notions into code? Can this process be automated?

In this paper we will try to answer the above questions. The paper proposes a notation for user stories [2] and notions that allows for easy translation into design level models and code. It also defines appropriate transformations. It is also argued that keeping the transformation mappings allows for more agility in treating the constantly changing user requirements.

2 Communicating the Users with the Developers

One of the fundamental practices of agile software development is close and constant cooperation of developers with the users. We can call such relations as "a cooperative game of invention and communication" [3]. However, we still have to bear in mind that clients and developers have their own backgrounds and their own points of view. A good way of communicating the users with the developers is to tell stories. Though, while most people like to listen to stories, only few are able in telling them well. Let's now ask a question in the context of software development: what would users and developers require from the stories? Gathering answers to this question would allow us to design suitable notation for the stories and determine the way they can be used in the development lifecycle.

Basically, the user wants to hear from the developer a story about "how the system shall work". The story should be communicated in a common language using simple sentences. The story should be a "real story" – with a starting event and with a "happy" or "sad" ending. These sentences should use well defined notions from the user's domain vocabulary. The notion definitions should be easily accessible when needed (not buried somewhere in other stories). The story should not use any special keywords or formal constructs. We should be able to group several stories that lead to a single goal from the user's point of view [4]. The users are usually not good in writing stories. They need someone that would listen to their stories and then write them down in some possibly standard way. Then, they can read the stories, judge them and suggest corrections.

The developers need stories to determine single units of the system's behavior to be developed. For them, it would be ideal if the stories were written as temporal sequences of interactions between roles and the developed system. Notions used in the stories should be easily transformable into design and code constructs (classes, interfaces, etc.) Stories should be formed of sentences that

allow for easy translation into sequences of code instructions. Moreover, when a story changes, it should be easy to trace the change into code. Developers that design systems and write code are often reluctant to write stories. However, they can easily verify quality of the stories from the design point of view. They are good at spotting inconsistencies, ambiguities and generally "holes" in the stories.

To write good stories we thus need good "story writers". We need to find them in our user or development team (and it seems not to be that easy). We also need to give them a "toolbox" which is necessary in order to establish a proper communication path. This toolbox should contain a notation for stories and tools to write them down. Developers would also be happy with a tool that could support translation of the stories into code.

3 Notation and Tools for Coherent User Stories

How to write a good story that would suit both the users and developers? Good stories, as they were written for centuries (see Orzeszkowa's novel), constitute a balance between describing the sequence of events and describing the environment. Good stories are also written in a coherent style.

When writing stories for a software system, we define a dialog between a user and the developed system (see [5] for an insight on such essential dialog). A story is a sequence of interactions between them. A good style here would be to describe these interactions with the simplest possible sentences, like:

- Student enters the semester.
- Teacher accepts the current marks.
- System assigns the student to the new semester.

These simplest sentences contain just the bare minimum for a full sentence: a subject, a verb, and one or two objects. What else do we need to tell a story? Well, of course we need some explanation of what do all the terms used in these sentences mean. Sometimes this may be quite obvious, but in many cases it is crucial to define them. For instance: the semester. One might think that it is simply a number between 1 and 10 denoting the level of studies in a five-year MSc program (like: I'm on the 5th semester). But maybe we are wrong, maybe it is the current academic half-year (like: the current semester is winter 2004/2005)?

Subject-verb-object (SVO) sentences are good at telling the sequence of events, but they are not appropriate for describing the environment of our story. So, what should we do? Should we allow for "SVO's with descriptions"? Experience shows that this is not a good idea. Writers are then tempted to write something like:

- Student enters the semester where the semester is a number between 1 and 10 denoting the level od studies.
  and somewhere else:
- Dean accepts the semester for the new student, where the semester is a number denoting the current student's status.

These two sentences are usually written in separate stories by separate writers or there is some period of time between writing them. The problem is – which

of these slightly different definitions is correct? Another problem is – how to find out that we in fact have two different definitions of the same notion? The solution is quite obvious: write only SVO sentences to describe the sequence of events and keep the notion descriptions in a separate vocabulary. This leads to significant improvement in coherence:

- Student enters the semester.
- Dean accepts the semester for the new student.
  where:
- Semester – number between 1 and 10 denoting the level of studies and the current student's status.

Having a notation for sentences we are now ready to tell full stories. These stories should form complete sequences of events. They should start with an initial interaction from a user and end with a final goal that gives the user a significant value. Requirements that contain such stories are perfectly suited for incremental development. They carry value for the users, and at the same time, they can be treated by the developers as atomic pieces of development. In every iteration we can now create increments based on stories like (see also Fig. 1, 2):

1. Student wants to select lectures.
2. System shows a list of possible lectures.
3. Student selects a lecture from the list of possible lectures.
4. System assigns the lecture to the student.

While writing the stories we constantly extend our vocabulary of notions. We describe all the sentence objects that might cause ambiguity when developing the stories. We also define relations between the notions. This gives us a static map of the "user's territory". We can write individual notions on index cards or we can draw simple class diagrams like the one shown on Figure 1 (see also [6]), consistent with Agile Modeling practices (like: Create Simple Content; [7]). These diagrams additionally extend and clarify our definition of semester:

- Semester – contains a number between 1 and 10 denoting the level of studies and the current student's status; has an associated **course** and a **list of possible lectures**.
  where:
- Course – contains several **semesters**; courses are taken by the **students**.
- List of possible lectures – a list of **lectures** associated with a given **semester**.

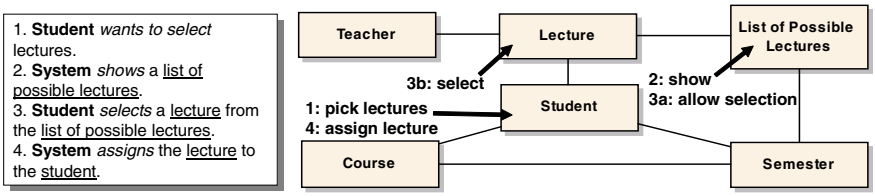

Fig. 1. Navigation through the domain for the "select lecture" story

User stories written with the above notation have two significant characteristics: they can be easily kept coherent and they can be easily transformed into design and code. Coherence of such stories lies in the fact that all of them are based on the same domain description. Notion definitions form a map of the territory that "glues together" functionality which sometimes seems totally independent. Stories only navigate through this static map giving it the necessary element of functionality, as illustrated on Figures 1 and 2. It verifies that all the notions used in our two stories are properly used. Note, that we have actually discovered that there are some inconsistencies in the stories. The story writer forgot that the system needs to determine the semester, before the student selects a lecture and that the dean should select the semester when adding a new lecture.

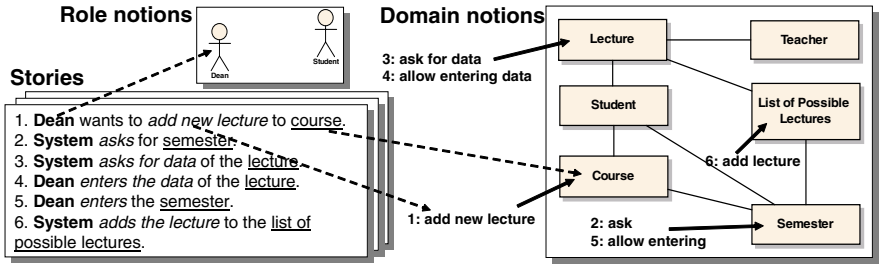

Fig. 2. Model of requirements based on "stories with notions"

Finding such inconsistencies is a difficult task having a typically sized system. We have hundreds of stories and tenths of notions to verify. It seems obvious that we need a tool to support our efforts. We can use the simplest possible "tool" – index cards with notions arranged on a wall. They can be manipulated easily and can be used to "play stories" as illustrated above. A step further would be to have this repository of notions and stories managed with an automated tool. This tool would allow us to organize sentences into stories, and hyperlink subjects, verbs and objects to appropriate elements of the vocabulary (see [8] and [9] for an example and a more detailed idea of such a tool). The stories and the vocabulary organized through hyperlinks, form in fact a model of requirements (Figure 2). We can call this model – "stories with notions". In the model, sentence subjects are linked to notions denoting roles for the system. Sentence objects have links with individual notions of the problem domain, and verbs denote operations on these notions. A model organized in a tool as described above has an important characteristic – it is ready to be transformed into a code model.

4 Getting from SVO User Stories to Code

Users write stories to tell developers what they need. Developers write stories to clarify that they have understood users correctly. Users and developers write

stories together to make sure that the final system will be correct (i.e. will satisfy the **real** needs of the user). This means, that we can in fact have various kinds of stories.

- initial user stories – simple statements from the users that reflect their wishes ("user wishes"),
- clarified user stories – more elaborated stories that contain more details about the functionality of the developed system,
- test stories – detailed stories with added test data, written for the purpose of acceptance testing.

All of these kinds of stories can be written using the SVO format. The initial stories can be written as just one to four SVO sentences. The clarified stories usually add to the initial stories more sentences (more details of the functionality) and add notions (details of the problem domain). These clarified stories can be "adorned" with test data to form test stories.

While the initial stories can jump-start elicitation of user needs, the more detailed stories are the starting point for all the development efforts. This leads us to defining a simple development cycle that uses SVO stories. A single iteration in such a lifecycle might look (in a simplified form) as follows (Fig. 3).

- The users write initial stories.
- The users meet with developers during a story writing session. Together they write clarified stories with notions.
- The developers translate the clarified stories into code. Coding is supported by class model derived from notions and sequence model derived from stories. During development, the stories are clarified with the users whenever necessary. Test stories are also written.
- Developers make sure that test stories are fulfilled and hand the system to the users.
- Users verify the system and possibly write corrected initial stories. They also write new initial stories. The lifecycle loop closes.

The simplest and possibly most efficient way to capture SVO stories and notions during story writing sessions are index cards. However, having a large set of stories and associated notions it seems worthwile to use an automated tool to support story clarification during development. Such a tool (mentioned in the previous section) can help organizing stories and keeping the overall model coherent.

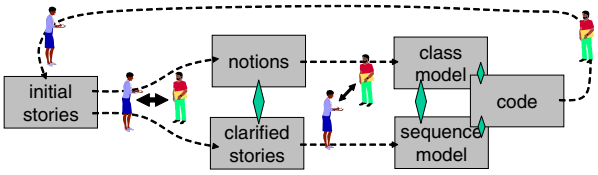

Fig. 3. Software lifecycle involving SVO stories and notions

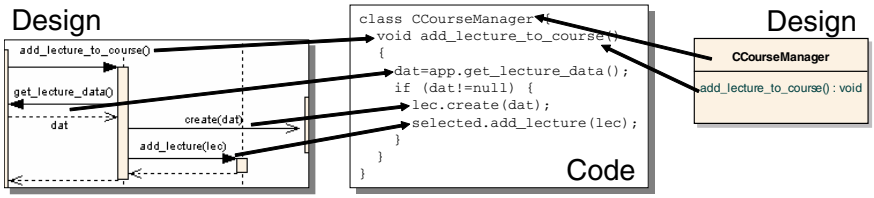

Fig. 4. Visual design models adding important abstraction level to code

As it can be noted on Figure 3, the overall development effort has been split into two groups of human activities. The first group involves translating the actual stories (treated as sequences of events) into sequences of instructions in code. The second group translates the domain notions into static code elements (classes). It is important to note that this translation can be done with the sole use of a typical programming environment only for the most trivial systems. Average systems are complex enough to necessitate some way of taming this complexity. In our approach, this taming is done through UML [10] class and sequence diagrams. These diagrams show the structure of code and its dynamics in a visual form (see Fig. 4). In most cases, diagrams hand-drawn on a white-board suffice. It can be also very beneficial to use an automated CASE tool (chosen from several on the market), integrated into our programming environment. This integration is essential, as only then it relieves us from the burden of actually synchronizing the pictures with code, and gives significant advantage over hand drawn pictures.

It has to be stressed that the use of UML CASE tools has to be done with great care. UML 2 has thirteen different types of diagrams. Extensive use of these diagrams might cause a severe "UML fever" (see [11] for an excellent survey of possible fevers). It seems from the current experience (several "clinical tests" were made) that applying the above lifecycle with class and sequence diagrams does not cause the UML fever. This is supposedly due to the fact that the diagrams in the lifecycle are drawn for a very specific purpose, which is to support structuring code in a clear manner. With CASE tool support, visual (graphical) documentation is created automatically while coding, similarly to using eg. JavaDoc. This makes creating additional heavy documentation completely unnecessary. At the same time it supplies the developers instantly with an additional level of abstraction that enhances comprehension of code.

The design model based on class and sequence diagrams has one more advantage. It can be linked directly with the requirements model based on SVO stories with notions. Keeping these links is important when handling changes in user needs. When SVO stories or notions change (and change the associated acceptance tests) we have direct visual pointers to places in code that need to be updated. These links can be kept simply by assigning appropriate index cards to appropriate hand drawn diagrams. SVO story cards are linked to sequence diagrams, and notion cards are linked to class diagrams (see. Fig. 5). With CASE

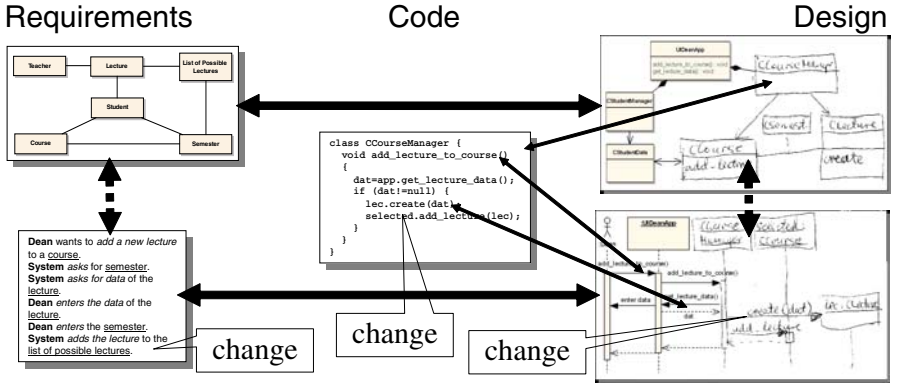

Fig. 5. Links between requirements and design pointing unambiguously to code

tool support, these links can be managed automatically thus supporting the human efforts. Keeping links between SVO stories and design diagrams can be treated as a version of Agile MDA [12]. It is a skilled (human-led) transformation between an inclusive model (user stories + notions) and the design model (class and sequence diagrams). The current approach can also be compared to Property-Driven Development [13], however, here the sequence diagrams are developed as part of the design model, not the requirements model (SVO stories are used instead).

It can be noted that the presented lifecycle uses practices already present in XP [14] and FDD [15]. SVO stories can be best compared to XP's user stories, with added SVO notation. The concept of structuring the initial user needs can be found in FDD, where features (and feature sets) have a very precise notation. The story writing session is closely related to the XP's planning game. Translating stories into design and then to code can be found as two major activities of FDD (design by feature, build by feature). Here, they are applied not to features as in FDD, but to SVO stories. Comparison with FDD can also show that actually the notion model is a simplified version of the "overall object model" (without attributes and operations). The two UML models used to design and document code are directly taken from FDD. UML models can also be applied with success in XP (see eg. [16]).

"Clinical tests" in a student lab project were made to verify that the method cures the UML fever. The students trained in UML were formed in groups of around 12 (around 7 groups in a year). These groups were assigned to develop a system during a one-semester lab. For three consecutive years, the lifecycle used during the lab changed from iterative, use case [4] driven with extensive UML models, through story-based with FDD [15] lifecycle, to SVO story-based. The SVO-based lifecycle seemed to be best suited for the less experienced developers like students. It resulted with higher interaction between the students and the "users" (i.e. the tutors) and much clearer code (see [16]). The final systems were more functional and more compatible with the real needs of the client.

5 Conclusions

Author's experience shows that catching the UML fever in software development is quite common. Most of the consulted development organizations were initially very eager to use automatic CASE tools that support UML notation and generate code automatically (well, almost automatically...). These tools looked like "silver bullets" for all their problems. They promised shorter (maybe one day long?) development cycles. Unfortunately, applying CASE tools in a documentation heavy environment resulted in a "look how much documentation we can now produce" syndrome. The organizations that wanted to change their development practices got bogged down in creating detailed analytical models supported by heavy architectural studies. This resulted in rejecting the tools as adding more work and returning to previous practices.

The approach presented in this paper seems to offer a cure for organizations caught by the UML fever. It offers a lightweight process, where UML diagrams are treated with great care. Tools are used only to support the actual development of working code by giving instant design level models. These models are very distinct from heavy documentation and give the advantage of having a higher level of abstraction (like XP's metaphors, and simple design diagrams). Moreover, these design models can be directly linked to the requirements models. Clear separation of structure from dynamics results in better communication among developers and between developers and users. This separation is done already on the requirements level and thus allows for clear distinction of activities that lead to implementing the domain structure (notions) from those that implement the system's dynamics (SVO stories). This distinction also supports invention and discovery. Many valuable notions can be discovered when SVO stories are used. It can be argued that supporting human activities with a clear path from constantly changing user needs to code, and with some simple transformation tools, could lead in the future to real reduction in development times (one day cycles?). However, this is still to come...

References

1. Breitman, K., Leite, J.: Managing user stories. In: International Workshop on Time-Constrained Requirements Engineering 2002 (TCRE'02), http://www.enel.ucalgary.ca/tcre02/ (2002)
2. Cohn, M.: User Stories Applied. Addison-Wesley (2004)
3. Cockburn, A.: Agile Software Development. Addison-Wesley (2002)
4. Cockburn, A.: Structuring use cases with goals. Journal of Object-Oriented Programming **5** (1997) 56–62
5. Constantine, L.L.: What do users want? Engineering usability into software. Windows Tech Journal (1995) revised in 2000, http://www.foruse.com/articles/whatusers.htm.
6. Ambler, S.W.: Agile data modeling. http://www.agiledata.org/essays/agileDataModeling.html (2005)
7. Ambler, S.W.: Agile Modeling (AM) practices v2. http://www.agilemodeling.com/practices.htm (2005)

8. Gryczon, P., Stańczuk, P.: Obiektowy system konstrukcji scenariuszy przypadków uzycia (Object-oriented use case scenario construction system). Master's thesis, Warsaw University of Technology (2002)
9. Śmiałek, M.: Profile suite for model transformations on the computation independent level. Lecture Notes on Computer Science **3297** (2005) 269–272
10. Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman (2000)
11. Bell, A.E.: Death by UML fever. Queue **2** (2004) 72–80
12. Ambler, S.W.: A roadmap for Agile MDA. http://www.agilemodeling.com/ essays/agileMDA.htm (2005)
13. Baumeister, H., Knapp, A., Wirsing, M.: Property-driven development. In Cuellar, J.R., Liu, Z., eds.: Proc. 2nd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM'04), IEEE Computer Society Press (2004)
14. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
15. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR (2002)
16. Astels, D.: Refactoring with UML. In: XP 2002, The Third International Conference on eXtreme Programming. (2002) http://www.xp2003.org/xp2002/.