The Many Lives of an Agile Story: Design Processes, Design Products, and Understandings in a Large-Scale Agile Development Project

Aaron Read
Center for Collaboration Science
University of Nebraska Omaha
aread@unomaha.edu

Robert O. Briggs
MIS Department
San Diego State University
rbriggs@mail.sdsu.edu

Abstract

In Agile Software Development (ASD), stakeholders use stories to stimulate conversations that create and convey understanding of software requirements. Some authors have argued that ASD methods have limited applicability to large-scale projects because agile stories are not sufficient to capture the complexities of up-front design. This paper reports a 2.5-year field study of how an ASD team for a complex software system adapted the user story concept and the Scrum approach. The team sought to create a convention for representing agile stories which could capture the complexities of the system requirements without burdening the team with unneeded documentation. They developed eight different ways to represent a story. The core representation of the approach was called a HyperEpic, *a structured collection of closely-related HyperStories. HyperEpics required 90-99% fewer words than conventional specifications. Because of their dense form, Hyper-epics were not useful for other phases in the design/build processes. The team evolved a design/build work practice that proceeded in stages. In each stage, stories underwent a one or more transformations. Each transformation represented stories differently to create varied kinds of understandings among different stakeholder sets. The team was able to gain the benefits of ASD – faster development cycles, less documentation, rapid adaptation to insights and conditions.*

1. Introduction

While the use of Agile Software Development (ASD) is on the rise, and ASD teams are seeing faster times to market, ASD methods have not generally been considered appropriate for large-scale complex projects [1]. One goal of agile methodologies is to improve efficiency by reducing the cognitive burden of capturing and updating complex system specifications. In an environment where requirements change quickly, exhaustive specification documents would have to be repeatedly reworked or scrapped. In Agile methods, developers do not create exhaustive designs before they start building the system. Rather, they design only a small part of the system at a time, and do so only just prior to the time when that small part will be built.

Agile requirements modeling, the use of low-fidelity/low formality requirements modeling methods, has emerged as an effort to introduce requirements modeling in a way that still achieves efficiencies [2]. In ASD, requirements are often captured in the form of stories, which are one-to-two sentence descriptions of a user's interaction the system [3, 4]. The purpose of an agile story is to stimulate face-to-face conversation among system stakeholders to create understandings about why a capability is valuable and how it could be realized. Face-to-face communication among small groups can be rich, efficient, and accurate compared to static documentation, which that may become outdated as understanding grows and concepts change over time.

As projects grows larger, however, their complexity increases with the number of requirements that must be de-conflicted, the number and diversity of stakeholder interests that must be accommodated, and the number of changes that must be managed [5]. The need to understand and analyze the design of software becomes greater as the size and complexity of the project increases. Face-to-face conversation among stakeholders becomes increasingly difficult, and becomes a less-and-less effective way to create and convey clear understandings about the system. ASD typically does not use formal requirements modeling and documentation, which makes it difficult to analyze large collections of requirements for completeness, consistency and conflicts, and provides little basis for evaluating the potential impact of changes to requirements [6, 7]. Lacking formal documentation, project knowledge rests only in the minds of the people who participate in the discussions that gave rise to it. It can therefore be difficult to transfer project knowledge to a large team. Thus, ASD teams working on large scale projects struggle to balance the need for precise, verifiable detailed designs with

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.684

the need to minimize the cognitive load and wasted effort that can accompany detailed specifications.

This paper reports a 2.5 year exploratory study of that struggle in a large-scale ASD project in which the authors played several active roles, among them customer, product owner, user, storymaster, and test master. Throughout the life of the project, team members tried to devise a convention for representing agile stories that would be sufficiently parsimonious and timely that undue effort was not wasted in their creation, yet sufficiently detailed that the nuances and interdependencies among the requirements could be captured and conveyed.

In the end, the team developed eight different conventions for representing a story. They evolved a design/build work practice during which stories underwent a sequence of transformations. Each transformation represented stories at a different level of abstraction or detail, or in a different format to create different kinds of understandings among different stakeholder sets.

In the following sections, we present our research methods and describe the project context. We then present the stages of the design/build work practice that emerged. For each stage we discuss the work process, the ways stories were transformed and represented, and the understandings the transformations were meant to support. We discuss the merits and disadvantages of the emergent design/build approach, and propose future research.

2. Methods

This research could be characterized as Action Research, because the authors participated actively in the design/build work practices under study [8]. This work could be characterized as Design Science [9] because its objects of inquiry are design processes and design products, because it was informed by the academic literature on ASD, and because it contributes this study back to the knowledge base. This study could be characterized as exploratory research [10] because it seeks to discover and describe phenomena in the context where they manifest. Accordingly, data collection methods for this study were primarily qualitative. We recorded daily observations, events, conversations, and interviews in our field notes. We collected and archived project documents and e-mail exchanged. After each design/build cycle, we conducted a qualitative retrospective about the advantages and limitations of the current work practices. During each retrospective, we conducted design discussions with key stakeholders about how to improve the design/build work practice, and how to improve story representation formats for the next cycle.

3. The Project

A group of civilian, military, and academic stakeholders (of which we were members) from the U.S. and the Netherlands agreed to build a new rapid development environment for collaborative software applications called *ActionCenters.*

Drawing on experience with earlier prototypes, we anticipated the project would require approximately 500,000 to 1,000,000 lines of new code.

The team had a budget of $3,600,000 USD, which they anticipated would only be sufficient to build the core functionality of the proposed system. Further funding would depend heavily on the degree to which the funding agencies could derive value from first round of development. A two-week project kickoff meeting was held in early January of 2009 in the U.S., during which a preliminary architecture for the system was defined. A second two week session was held in March, 2009 in the Netherlands, during which most architectural issues were settled, and a roadmap was created for the creation of key capabilities.

The ActionCenters project team consisted of a one Problem Owner (PO), a System Architect (SA), four full-time senior developers, two part time developers, a UI designer (UI), and a Story Master (SM). Each professional team member teamed with an intern/mentee. A third of the way through the project, the testing team formed consisting of a Test Lead (TL) and two testers.

The PO had been involved in the design and development of an earlier full-scale thick-client proof-of-concept prototype of the system. None of the prototype code could be used in the field-grade version of the system because the team decided to implement the new system on a thin-client platform. None of the developers of the prototype system was involved in developing the production version.

Full formal specifications for the proof-of-concept prototype system existed. The PO and the team on the original prototype development team had spent a year writing those specifications before starting to code the system. Those specifications could not be reused, however, because they were too closely tied to the original development platform. The team found that it was not possible to reverse engineer the requirements and constraints from the detailed system specifications. The PO had a detailed understanding of what the capabilities and constraints of the system should be. Those requirements remained stable over the life of the project.

During the project, the development team logged 11,084 hours of work, and a comparable level of effort was devoted to design and test efforts. As of the writing of this paper, the core functionality of the *ActionCenters* platform has been completed. Collaboration Engineers can use it to design, publish, and run custom collaborative environments with a wide range of configurations.

3.1 Selection of an Agile Software Development Methodology. At the beginning of the project, only the PO knew the system requirements. The PO had other duties in addition to the project, and so could not devote full time to communicating the requirements to the development team, and was not willing or able to spend another year writing new specifications. The PO and development team therefore decided to use an Agile development approach for the project, hoping to eliminate the cognitive load associated with writing formal specifications. The PO also expected to take advantage of the flexibility of developing software iteratively. None of the development team had previous experience with Agile methodologies.

The development team established work practices based on the Scrum (Highsmith and Cockburn 2001). The SA also served as the Scrum Master. The team agreed to develop software in 30-day iterations called "sprints". The PO, the scrum master, and SM would work together to decide which stories should be implemented in each iteration. Each sprint would begin with a one-to-two day planning meeting where the stakeholders would review what had been accomplished and learned in the previous sprint. Then developers would estimate the level of effort required for stories to be built in the next cycle. Developers made their estimates in terms of story points [4]. A story point roughly equated to an ideal half a day of work for a single developer. During the 2.5 years, the team executed 2,771 story points.

Stories were managed in a widely used commercial system called VersionOne. During each sprint, developers held a daily 15 minute stand-up meeting to report what they had achieved the previous day, what they would achieve that day, and what barriers they faced.

3.2 The Challenge. The PO was co-located with the development team at a university, but was constantly pressed with other duties, and so could only spend about ¼ of his time on the project. During the month before the technical team began to write code, the PO and SM met several times to write stories for the development team. Lacking experience, the PO and SM wrote stories were short, vague and ambiguous, such as, "*Login to system*," (note that this example says nothing about who should be allowed to login under what conditions, nor about what access controls should be in place for login), and "*We should be able to paste agenda items onto a project.*" (note also several missing details).

During the first sprint planning session, the development team asked few questions about the stories, but estimated the level of effort for the stories and committed to develop them over the course of the sprint. During the sprint, being inexperienced with agile methods, the developers and PO did not discuss what the stories meant, nor how they should be implemented. At the end of the first sprint, the PO was disappointed that many of the features did not function as intended, and some, although computationally correct, could not be used for their intended purpose. A frustrated programmer said, "The story didn't say I should do that, so I didn't do it."

For the next sprint, the PO tried to write formal specifications, but found that the level of effort required would far exceed the available time. The team concluded that the project would fail if they could not find a faster way to communicate about the requirements. They set out to find a way to write stories that would convey sufficient detail about requirements that the team could develop satisfactory code, while requiring less cognitive effort to create and maintain than formal specifications.

3.3 Toward a Solution. Over the next several sprints, the PO, the SA, and the SM, with input from all other members of the team, devised, piloted, and rejected several story formats, before conceiving on an approach they called HyperEpics (Table 1).

A HyperEpic was a structured collection of closely related HyperStories. The HyperStory was similar to a conventional agile story, but it followed a five-part structured format:

> In some *place*, a person in some *role* takes some *action*, for some *goal*, which produces some *effect*.

For example,

STORY: Delete Activity

In the ActionCenter Builder, a collaboration engineer deletes an activity from the Agenda to eliminate it from the work practice. The activity disappears from the ActionCenter Builder and appears in the recycle bin.

END STORY

Place refers to the aspect of the system where the story takes place (e.g. the ActionCenter Builder). *User Role* refers to the type of user who can execute the action (e.g. a collaboration engineer). *Action* refers to what the user will do in the system (e.g. delete an activity). *Goal* refers to the reason the user wants to execute the action (e.g. to eliminate an activity from a work practice). *Effect* states what should happen when the user takes the action (e.g. an activity disappears from the agenda and appears in the recycle bin).

Any aspect of a HyperStory that would be repeated frequently in other stories was abstracted into a reusable parameter. Any objects or events for which a story would be repeated were likewise abstracted to reusable parameters with multiple values. Parameters were inserted in brackets into a HyperStory, for example:

STORY: Delete [Agenda Item]

In a [CACE Editor] a [user] deletes an [agenda item] from the agenda in order to eliminate it from the work practice. The [agenda item] disappears from all [CACE Editor]s and appears in the recycle bin.

END STORY

PARAMETERS

[CACE Editor]: Agenda editor; FPM Editor; ActionCenter Builder

[user]: Collaboration Engineer; Guest]

[agenda item]: Phase; Activity

END PARAMETERS

Closely related HyperStories that shared parameters were typically developed in parallel as a HyperEpic like that illustrated in Table 1. A unified list of parameter values from all stories appears at the end of the HyperEpic. HyperEpics were typically composed of five to fifteen stories.

The approach substantially reduced cognitive load and time-on-task for design documentation, while providing significantly more details than conventional stories. The HyperEpic format required 90-99% fewer words than were needed for formal specifications. The two HyperStories in Table 1, for example took the place of 144 conventional specifications. Further, the HyperEpics were more-easily maintained than formal specifications, because a single revision to a parameter would propagate to all stories where the parameter appeared. Because the logic of a story could be considered separately from the values of its parameters, the format made it easier to validate consistency and completeness of requirements.

Because of their dense, compact form, however, HyperEpics were not useful for many phases of the project. Programmers, for example, found it difficult to hunt through a HyperEpic to find the particular story on which they were to work, and then to work back-and-forth between the story and the parameter list, which could be several pages away. Further, many parameterized HyperStories, although they were compact, were, nonetheless, too large to be executed in a single sprint, and so were not a useful unit of analysis for estimating development effort for a sprint. The team therefore evolved a design/build work practice that proceeded in stages. In each stage, stories underwent a one or more transformations to produce artifacts that would be useful for the goals of the phase.

In the following section, we describe the phases of the work practice. We present the understandings that each phase was meant to create, and explain the transformation(s) that stories underwent so those understandings could be reached. We provide examples of each story format.

4. The HyperEpics Design/Build Work Practice

The HyperEpics work practice proceeded in three concurrent cycles: The design cycle, the build cycle, and the test cycle. During a given sprint, the design team would design the capabilities that were to be built in the following sprint; the developers would build the capabilities that had been designed in the previous sprint, and the testers would test the capabilities that had been developed in the previous sprint. A story was born and transformed during the design cycle, further transformed several times during the build cycle, and transformed again for the test cycle. Stories were initially created during the Preliminary Design phase.

4.1 Preliminary design. The purpose of the preliminary phase was to understand the goals of the users of proposed system capabilities, and to design high-level interface/user experience (UI/UX) and technical approaches for providing those capabilities. The system under development was so complex that it presented some daunting (UI/UX) challenges. One interface, for example, was sufficiently challenging that it took 3 months of repeated efforts before the team found an elegant solution that both supported the complexity of users' needs, and presented a simple-to-use user interface.

The team found it useful to develop UI/UX and technical implementation solutions simultaneously, because it was frequently the case that two equally useful UI/UX designs would require very different levels of development effort. Further, the technical people often provided inspiration for UI/UX solutions, and the UI/UX people often provided inspiration for technical solutions.

Table 1. Example of a small HyperEpic. A HyperEpic is a structured collection of closely related HyperStories. A HyperStory follows a structure format: In some *place,* a person in some *role* takes some *action,* for some *goal,* which produces some *effect.* Any aspect of a story that would be repeated in other stories, and any objects or events for which a story would be repeated are abstracted to reusable parameters. Parameters appear in brackets in a HyperStory. A list of parameter values appears at the end of the HyperEpic. HyperStories in the same HyperEpic typically share one or more parameters. The two HyperStories shown here take the place of 144 conventional specifications

EPIC: *Open and Close Element editors*

STORY: *Open Element Editor*

In [a CASE Editor], a Collaboration Engineer [opens] the element editor for an [object] to configure the features of that [object]. The element editor for that [object] appears on a new tab in the CACE editor pane. Fields with values set appear with those values. Fields with no values set appear empty.

BUSINESS RULES:

The first time an element editor for an [object] is opened, default values will be set for all fields

END RULES

END STORY

STORY: *Close Element Editor*

In the CACE, A Collaboration Engineer [closes] an [object] element editor to hide the configurable features of that [object] to reduce visual load. The element editor for the object closes. The values of all fields in the element editor are saved to the server.

BUSINESS RULES

A user cannot not close an element editor until the all its fields have displayed

END RULES

END STORY

PARAMETERS:

[a CACE Editor] *Explorer Tree; ActionCenter Builder; Screen Editor; Tool Editor*

[closes] *clicks the x on the element editor tab; right clicks element editor tab and selects close; clicks the close button on the element editor*

[object] *Project; Phase; Activity; Role; Screen; Tool; Component; Control*

[opens] *Double-clicks; Right-clicks and selects "Edit"; Clicks the File menu and selects "Edit"*

END PARAMETERS

END EPIC

Preliminary design processes were conducted in weekly and bi-weekly face-to-face meetings that lasted four to eight hours. Typically, these meetings involved the PO, UI, SA, and SM. The inputs to this phase were usually a) a system aspect from the development roadmap; and b) oral statements of user goals. PO and UI typically made a series of rapid sketches of possible user interface solutions on a whiteboard to support the UI/UX discussions. PO and SA sometimes sketched data models on the whiteboard to support discussion of approaches for technical implementation. All participants contributed to all aspects of the discussion. When the group reached consensus on an approach, some member would turn to the SM and say, "Catch a story title on this." This was the first format for a story.

During the design session, story titles were captured as brief, sometimes cryptic phrase or sentence that with just enough information to remind the PO and SM about the nature of the solution to which the group had agreed, for example:

"Lobby roles for newly created accounts"
"Alphabetize the Explore"
"FFFIBBU on Population Rules"

The group would discuss the wording of the story title, take a picture of whiteboard diagrams, and then move on to the next design challenge.

At the end of the meeting, as a memory aid, some of the more complex story titles would be transformed to a somewhat more structured format, similar to, but not as complete as that of a HyperStory, for example, the story title:

> *"Open Activity Element Editor"*

might be elaborated to read:

> "*in the CACE tool, A collaboration engineer opens an Activity Element Editor with the mouse or keyboard.*"

Elaborated Story Titles could include some or all of the HyperStory elements, Place + User Role + Action + Goal + Effect. Most did not include the goal, however, and they never included parameters.

The final design products for the Preliminary Design Phase were Story Titles and photographs of whiteboard sketches. The story titles represented design commitments agreed to between the PO, SA, and UI. The elaborated title format helped to assure that the important details of the discussion were captured. These products were well suited to capturing the understandings achieved in the design phase because they required very little effort to create, which meant that capturing them did not disrupt the design process. They had a limited shelf life, however. If they were not taken into a Detailed Design session within a day or two, the stakeholders would forget the nuances and details of the design approaches to which they had agreed. On several occasions, when the team waited too long to execute the next phase, they had to repeat the preliminary design for some aspect of the system.

4.2. Detailed Design. The goal of the detailed design phase was to create complete and consistent catalog of the functionality required for system capabilities being designed. The goal was not to create complete specifications. Rather, it was to create a complete set of expectations that could be discussed with the development team. The inputs to this phase were the story titles and whiteboard sketches produced during Preliminary Design.

In this phase, The PO and SM would transform Story Titles into fully parameterized HyperEpics (Table 1). HyperEpics were high-level narrative of the user's experience with the capability. A HyperEpics contained a collection of related titled, parameterized stories, the list of parameters the stories shared, and a list of business rules that clarified the expected behavior of the capability. Story parameters represent, for example, lists of objects to which the story applied, or lists of events that would initiate the story's action. Business rules might include, for example, algorithms for calculations, or constraints on actions specified in the story.

The Detailed design Phase almost always exposed issues and opportunities that had not been considered during Preliminary Design, so HyperEpics typically contained many more stories than were represented by the story titles created in the previous phase.

During the Detailed Design phase, UI would often transform whiteboard sketches into two kinds of formal drawings: wire frames or pixel perfect comps. A wire frame diagram represented screen elements as simple shapes, showed their relative positions and proportions on the screen, and showed the wording that should be used, for example, on menus, buttons, and screen prompts. Pixel perfect comps showed exact representations of user interfaces down to the position and color of very pixel on the screen. Wire frames were faster to draw, but some user interfaces had so many nuanced details that comps were a faster way to communicate the requirements to the programmers. Wireframes, Comps, and whiteboard diagrams were attached as digital objects to the HyperEpics to which they pertained.

The final products of the Detailed Design Phase were HyperEpics with their associated Wireframes and Comps. The transformation of story titles into HyperEpics helped the PO and SM understand whether the design of a capability had matured sufficiently for the team to consider its development. HyperEpics also gave PO, SM, and SA a better understanding of the relationships, overlaps, and dependencies among capabilities; HyperStories that shared parameter sets frequently wound up sharing code.

4.3 Story Evaluation Phase The goal of the Story Evaluation phase was to understand the relative importance of each story for the success of the project, and the technical complexity of implementing it. The inputs to the Story Evaluation Phase were the HyperEpics and UI designs from the Detailed Design phase.

While HyperEpics were a very fast way to capture complete and consistent requirements, they were too dense and too complex to be evaluated as a unit. The Story Evaluation Phase therefore began with the SM transforming the HyperEpics into stand-alone HyperStories. Each story appeared with its own parameter set, and its own business rules, for example:

STORY: *Open Element Editor*

> *In [a CASE Editor], a Collaboration Engineer [opens] the element editor for an [object] to configure the features of that [object]. The element editor for that [object] appears on a new tab in the CACE editor pane. Fields with values set appear with those values displayed. Fields with no values set appear empty.*
>
> *BUSINESS RULES:*
>
> > *The first time an element editor for an [object] is opened, default values will be set for all fields*
>
> *END RULES*
>
> *PARAMETERS:*
>
> > *[a CACE Editor] Explorer Tree; ActionCenter Builder; Screen Editor; Tool Editor*
> >
> > *[opens] Double-clicks; Right-clicks and selects "Edit"; Clicks the File menu and selects "Edit"*
> >
> > *[object] Project; Phase; Activity; Role; Screen; Tool; Component; Control*
>
> *END PARAMETERS*

END STORY

One goal of breaking down the HyperEpics was to decompose them into estimateable chunks. Some of the resulting HyperStories were still too large to estimate. These would be split into separate stories, each with a different subset of the parameters. The SM entered all HyperStories into the backlog of the VersionOne Story Management system. When possible, the SM linked supporting artifacts, such as diagrams to serve as a reminder about proposed design details.

One week before the end of a sprint, PO and SA would transform further by adding priority and complexity ratings to each story. The PO would assign one of four qualitative priority ratings to each story. A story would be rated "Critical" if it was essential to the core capabilities of the system. It would be rated low if it was a convenience or cosmetic issue. The SA would assign one of four qualitative complexity ratings to each story (Low, Medium, High, Too High, Too many Questions). A story would be rated "Too High" if the SA believed it would take more than one sprint to execute the story. That story would be further decomposed into smaller stories. A story would be rated "Too Many Questions" if the SA found it too ambiguous to estimate. The SA and PO would discuss and annotate the story and the SA would reevaluate it.

Items with a Critical rating were placed at the top of the backlog in rank order of priority by the PO. A cutoff line was placed in the backlog to designate which stories that would be considered for development during the next sprint.

The final design products from the Story Evaluation Phase were Evaluated HyperStories - HyperStories and their associated diagrams rated on priority and complexity. Transforming the HyperEpics into Evaluated Stories helped the PO and SM understand the development trajectory for the system, and helped the SA understand the expectations of the PO for the next sprint, and the levels of effort that might be required.

4.4 Sprint Planning Meeting The Goal of the Sprint Planning Meeting was to understand what capabilities would be developed during the next sprint. Sprint Planning Meeting marked the end of one sprint and the beginning of the next. All project stakeholders participated in this event.

The meeting began with a retrospective during programmers would demonstrate story-by-story the new capabilities they had built in the previous sprint. New stories often emerged during the retrospective as bugs were discovered or as new opportunities were identified. These were added directly to the backlog. If the story were deemed sufficiently important to be addressed in the next sprint, it would be placed "above the line" at the top of the backlog and evaluated for priority and complexity on the spot. If it were not sufficiently critical, it would be placed unevaluated "below the line."

After the retrospective, the technical team would decide how many story points they could complete in the next sprint, taking into account number of points they had executed in the previous sprint, holidays, vacations, and other issues that might affect their productivity. Next, they focused on the backlog, working story-by-story from the top down to estimate the number of story points each story would require. First, the PO would explain a story to the other stakeholders, who would ask questions until they understood not only the story, but why the capability was needed. The team would discuss the technical details of how the story could be implemented.

Next, the development team members would estimate the number of points the story would require using the "Points Poker" technique. Each developer had a stack of numbered cards that approximated a Fibonacci sequence: 0, 1, 2, 3, 5, 8, 13, 20, 40. Each developer would select a card to representing the number of points that would be needed to execute the story. The rule was, if they believed a story couldn't be developed for a smaller number of points, they had to jump to the next number in the Fibonacci sequence. If, for example a story could not be completed in five points, the developer would have to estimate 8 points. On a signal, all developers would reveal their estimate cards simultaneously. If the cards matched, then that number of points would be assigned to a story. If the cards differed by no more than one number, then the highest number would be assigned to the story without discussion. If the cards differed by more than one number, then the person with the highest card would have to debate the rating with the person who displayed the lowest card. After the debate, a new round of voting would take place. This would continue until ratings were different by no more than one number. If a developer did not know enough to offer an informed estimate, they would abstain by displaying a question-mark card.

Often, the story-point debates revealed issues that had not yet been considered. The resulting conversations often led to better implementation solutions than had been devised in the design phase. Sometimes these debates gave rise to new stories that were entered into the backlog for further consideration. It was often the case that a HyperStory was too large to be completed in one sprint. Such stories would be split. The developers would commit to complete some of the parameters of the story during the current sprint; the rest would be put back into the backlog for future consideration. More often than not, after the vote, the SA, with the cooperation of the development team, would transform a HyperStory into a unparameterized, plain language story expressed in terms that made sense to them. They would add annotations to the story about issues and agreements that emerged from the discussion of the story. The HyperStory was typically retained as an annotation to the plain language story, and annotations on the Plain-language story often referred to parameters of the HyperStory. Plain-language stories were often as cryptic as the initial story titles had been, for example:

Edit Role in Element Editor
-include double click on role in Explorer [refers to HyperStory]
-include double click on role in ACB [refers to HyperStory
3 Story Points

The exercise of transforming the HyperStory into a plain-language story, however, appeared to help the developers internalize the concepts of the story. We observed that, during development, they would frequently return to the HyperStory to remind themselves of its details.

Once the development points had been assigned to a story, the PO would decide whether the story should be developed in the next sprint. If so, the estimated story would then be moved from the backlog onto the development list.

The design products of the Sprint Planning Meeting were Estimated HyperStories and Estimated Plain Language Stories, elaborated with annotations about issues and agreements made during the session, and often annotated with the original HyperStory. Transforming HyperStories into these formats helped all stakeholders understand the goals and products of the next sprint, and helped the PO understand which stories would yield the most value for the effort expended, and so how to shape the development trajectory. The Sprint Planning Meeting ended of one Design Cycle and the beginning of another.

4.5 Task Analysis Phase_ The Development Cycle began the day after the Sprint Planning Meeting with the Task Analysis Phase. The goals of the Task Analysis Phase were to assign stories to developers, and to estimate how many hours would be required to complete each story, so the team could track their progress toward completion of sprint commitments on a daily basis. SA and developers would meet to discuss which developers would work on which story. Each developer would then analyze each story into a set of tasks, and estimate the number of hours each task would take. For example:
Markup xml to reflect properties -- 4 hours

Add all the objects to that Role can have – 2 hours
Double-click action on the role node -- 2 hours

Add the element editor the existing role editor group --4 hours

Next, the developers met to identify dependencies among their tasks, so they could plan the order in which their tasks should be executed. The tasks and hours for each story were entered into the story management system. As they completed their tasks, they recorded the actual hours spent on the task, and so were able to project whether they were ahead of or behind schedule.

Sometimes this analysis revealed that the developers had over-estimated or under-estimated the story points during the Sprint Planning Meeting. Over time, however, it appeared that the positive and negative deviations tended to cancel each other out, so no serious difficulties emerged. All sprints were completed on time, and most sprints actually completed more work than that to which the team committed.

Transforming Stories into Estimated Tasks helped developers to understand how they were going to

Table 2. The Many Lives of an Agile Story. Design Processes, Design Products, and the Understandings They Facilitated Among Stakeholders in a Large Scale Agile Software Development Project

Design Processes	**Design Products**	**Understandings**
Preliminary Design	Story Titles UI/UX sketches	Goals of the users of propose system capabilities. High-level understandings of UI/UX and technical approaches
Detailed Design	HyperEpics	Detailed understandings of required functionality Completeness and consistency of proposed solutions
Story Evaluation	HyperStories Evaluated HyperStories	Relative importance of stories to project success Complexity of implementing stories
Sprint Planning	Estimated HyperStories Estimated Plain Language Stories	What capabilities would be developed in the next sprint. Technical implementation strategy for each story
Task Analysis	Estimated Tasks	Production scheduling constraints. Technical implementation tactics for each story.
Acceptance Testing	Acceptance Tests	The extent to which newly developed code instantiates stakeholder expectations and intentions
Regression Testing	Regression Tests	Robustness of existing code after new code is added.

realize the story as system capabilities, and how well they were maintaining their development schedule.

4.6 Acceptance Testing Phase The end of the Sprint Planning Meeting also marked the beginning of the Test Cycle, which included Acceptance Testing and Regression Testing. The goal of acceptance testing was to assure that the software developed matched the capabilities intended by the PO. Acceptance testing included stress testing to identify defects as early as possible in the life of the capability. The TM attended the sprint planning meeting to assure that he understood what functionality would be developed in the next sprint. Afterward, the TM would transform stories into acceptance test cases. Acceptance tests decomposed HyperStories into Action-effect pairs:

Action
-Double click a role in the Explorer Tree.
Expected Result
-The element editor for the object opens in the main window of the CACE tool.

Action
-Double Click a role in the ACB.
Expected Result
-The Element Editor for the object opens in the main window of the CACE Tool

Action
-Double Click on a role in the explorer with the role element editor already opened in a tab, but with another tab viewable.
-Expected Result
-The role element editor becomes the active tab.

Acceptance Tests were run manually by testers soon as a story was completed by the programmers. Pass/fail results of tests were reported back to programmers. If a test failed, the developer responsible for that story would have the opportunity to fix the problem before the sprint finished.

The products of the Acceptance Test phase were Acceptance Tests, bug reports, and tested code. The transformation of stories into Acceptance Tests helped testers understand how the system was supposed to work. Acceptance tests were represented in a highly explicit format to communicate clearly how the software was to function.

4.7 Regression Test Phase The goal of the Regression Test Phase was to assure that new code introduced into the system did not have an adverse effect on existing code. In the Regression Testing Phase, selected Acceptance Tests were Transformed into Regression tests. Over most of the life of the project, Regression tests were coded into an automated test harness that would execute the tests without human intervention, and provide a report to the test team on any tests that failed.

Regression tests were coded with keywords pertaining to the aspects of the system they tested. Testers could select a batch of regression tests to test just the aspects of the system that might have been affected by recent changes. As the set of regression tests grew, priority was given to tests that had not been run as frequently or as recently, or that had failed during a previous sprint.

The products of this phase were regression tests and tested code. Regression tests helped testing team members who did not attend sprint planning sessions to understand how best to test the system.

5. Analysis and Discussion

Table 1 summarizes the design process, the design products, and the understandings they facilitated in the HyperEpic approach to ASD. With this work practice, stakeholders in complex software development project were able to understand, design, build, and test, and manage design, development, and test cycles dealing with thousands of detailed requirements. Instead of using a single representation of a user story, as is the conventional method of documenting requirements in ASD [4], we use 8 different formats, each tailored to a support specific understandings in the design, development or testing phase. We presented these formats of these representations, and discussed transformations they underwent at each work practice in its current state.

Reflecting upon the differences between our work practice and story representation and the practice described in the ASD literature [4], we can assess advantages and disadvantages of our method in a project where complex design software is being developed:

Advantages:

- This approach to formatting, transforming and organizing stories recognizes several different stages of design, build, and test. Conducting the phases of design allowed us to tailor the formatting of user stories to fit the different tasks of different design phases
- We successfully eliminated some of the waste of using an ASD process in a complex environment: the obsolescence of stories. When design changes were likely to occur, we captured only short story titles, eliminating the wasted effort of prioritizing and placing stories on the backlog.
- When structure is advantageous, we have incorporated structure. With the use of HyperEpics, we reduced the cognitive load of establishing and verifying the completeness and consistency of a set of requirements.

Disadvantages:

- Some flexibility, which is a key feature of ASD, is lost when more stages of design are recognized. In our own experience, there were cases where many of the design phases were not necessary. For example, following a retrospective, user stories would be placed directly on the backlog. If the team had not recognized intuitively that this was appropriate, effort would have been wasted formatting the story for unnecessary design phases.
- This method may be overkill for projects with lower complexity and fewer design details.

6. Conclusion

We have presented a work practice which has several advantages over traditional user stories used in ASD. We feel that our discovery of different design stages and formats has the potential to generalize to many ASD projects. In future research we will focus on the forces which led us to utilize these different stages.

References

1. Boehm, B. and R. Turner, *Using Risk to Balance Agile and Plan-Driven Methods.* Computer, 2003. **36**(6): p. 57-66.
2. Ambler, S., *The object primer: Agile model-driven development with UML 2.0*. 2004: Cambridge University Press.
3. Beck, K. and M. Fowler, *Planning Extreme Programming*. 2000: Addison-Wesley Longman Publishing Co., Inc. 160
4. Cohn, M., *User Stories Applied: For Agile Software Development*. 2004: Addison Wesley Longman Publishing Co., Inc. .
5. Wood, R.E., *Task complexity: Definition of the construct.* Organizational Behavior and Human Decision Processes, 1986. **37**(1): p. 60-82.
6. Turk, D. and R. France. R, *Assumptions Underlying Agile Software-Development Processes.* Journal of Database Management, 2005. **16**(4): p. 62-87.
7. Ramesh, B., L. Cao, and R. Baskerville, *Agile Requirements Engineering Practices and Challenges: An Empirical Study.* Information Systems Journal, 2010. **20**: p. 449-480.
8. Argyris, C., R. Putnam, and D.M. Smith, *Action Science*. Josssey-Bass Social and Behavioral Science Series. 1985, San Francisco: Jossey-Bass Publishers.
9. Hevner, A. and S. Chatterjee, *Design Science Research: Looking to the Future*, in *Design Research in Information Systems*, A. Hevner and S. Chatterjee, Editors. 2010, Springer US: New York. p. 261-268.
10. Stebbins, R., *Exploratory research in the social sciences*. 2001: Sage Publications, Inc.