Using Storyotypes to Split Bloated XP Stories

Gerard Meszaros

ClearStream Consulting Inc., 3710– 205 – 5th Avenue S.W.

Calgary, Alberta Canada T2P 2V7

gerard@clrstream.com

Abstract. An ideal XP project is composed of stories defined by the customer that are of the right size and focus to plan and manage according to XP principles and practices. A story that is too large creates a variety of problems: it might not fit into a single iteration; there are a large number of tasks that must be coordinated; it can be too large to test adequately at the story/functional level; too much non-essential functionality is bundled early in development causing essential functionality to be deferred. Teams new to XP find managing the size of stories especially challenging because they lack the experience required to simplify and breakdown large stories. This experience report describes four heuristics (storyotypes) we have used on our XP projects to successfully manage the size of stories.

INTRODUCTION

At ClearStream Consulting, we have helped many clients learn how to apply eXtreme Programming (XP) on their projects. A common problem they face is getting the right granularity for their stories; most projects start off with "bloated stories" that later need to be split into smaller stories.

Teams that have experience using "use cases" find it particularly difficult because use cases can have many scenarios. These scenarios can vary greatly in business value and should not be included in a single "use case story".

To help these clients learn how to structure their stories, we have come up with a set of four "storyotypes". We ask them to identify which storyotypes a particular story-candidate exhibits and if it exhibits more than one, we have them discuss the value of splitting the story into smaller stories, ideally one for each storyotype.

The focus of this paper is to share our experiences with managing the size of stories within XP projects. We start by describing the problems in managing the story size. We then describe the four storyotypes we have encountered on information system projects and how they are used to mitigate these problems.

The Problem with Stories

To understand the problems that are generally experienced with story granularity, a quick review of the XP concept of a story is helpful. Stories were first described in [1] & [2]. The customer is responsible for defining the functionality of the system in short "stories" of one or two sentences. Each story should describe functionality that has real business value to the customer. From a planning perspective, the story is the unit of prioritization, scheduling, and progress tracking that is visible to the customer.

An XP project has frequent small releases, each of which contains a number of time-boxed iterations. Release planning involves scheduling one or more stories in a particular release, based on the priority and size of the story. The entire story must be finished within the release for which it is scheduled otherwise no value is delivered to the customer. A large story creates problems in three areas: Release Planning, Task Coordination, and Story Testing.

Release Planning

The first problem that large stories create for an XP team is in release planning. The larger the stories, the fewer will fit into a release. (Larger stories are also harder to estimate.) This gives the customer less flexibility to pick and choose what gets done. Too much functionality bundled into a single story will often squeeze out other equally important core functionality from early releases thus delaying a meaningful demo unnecessarily. If the stories remain too large throughout the project essential core functionality may be squeezed out (differed indefinitely), because the earlier bloated stories contained non-essential functionality that consumed development resources.

Task Coordination

Task coordination is the second area in which problems can arise. A large story either generates a larger number of tasks or larger tasks. We have found the integration of these tasks can be problematic.

Our XP projects typically do not require micro-management of tasks to the extent that detailed grouping and dependencies of the tasks do not have to be worked out as long as the stories are kept reasonably small. With larger stories, extra overhead must be incurred to orchestrate the sequencing of cohesive tasks to ensure that the team makes progress towards a common sub-goal at any one point in time within the iteration.

Story Testing

The third problem experienced is that the granularity of the story testing is too large. The customer is responsible for specifying and signing off on customer tests. As a story becomes larger, there must be more extensive testing to deal with all the

interactions of the functionality. These interactions are difficult for customers to test all at once. We have found the completeness of customer tests drops as the number of tests needed by a story exceeds 10 tests. Smaller stories tend to have more complete testing than larger stories.

Using Storyotypes to Split Stories

Splitting of stories is described [1] & [2] as one of the basic techniques of managing scope on XP projects. A story should meet the following criteria:

- *Each story should describe functionality that has real business value to the customer.*
- *The stories should not have any value if they are further subdivided.*
- *The functionality described in a single story should have the same importance to the customer.* That is, the relative priority should be the same.
- *The functionality should have the same level of certainty.* That is, if some functionality is completely understood and some needs to be discussed in more detail with the business, there should be at least two different stories because one is ready to be built now and the other is not.

Further guidelines are provided for the "bootstrap story" (the first story built; a special case on every project) in [3].

These guidelines help newcomers to XP, but they don't help them figure out how to make a story the right size. Those coming from a use case world have a tendency to want to use the functionality described by a use case as the basis for their stories. But use cases are the wrong granularity for stories. They are both too big and too small at the same time.

Use Cases are Too Small. Many use cases cannot be tested independently of other functionality. That is, while they might be executed independently, the results cannot be verified without using some other use case to inspect the state of the system. Or, the use case may depend on some other use case to set up the state of the system before it can be exercised.

Use Cases are Too Big. While there are many definitions of what constitutes a use case, most definitions agree that it includes all the possible ways a user can achieve some goal or desired outcome. Typically, a use case has several or many scenarios. Some of these scenarios are used very often (the "happy path" scenario and a few others) while others may be pathological cases that occur so rarely that it is not worth automating them. That is, they provide insufficient "business value" to justify the investment to automate them through software.

Usage Scenarios are Better but Not Enough. Use cases typically consist of several or many scenarios (the "alternate paths" through the use case) that describe how the use cases works with various prior states of the system. Each scenario can be considered a candidate for a separate story so that it can be prioritized independently of the other scenarios. To address the "Use cases are Too Small" problem, they often need to be combined with scenarios of other use cases to make a truly testable story. And even scenarios can be too big to build in a single release.

Four Storyotypes

To make it easier for new XP teams to come up with the right story granularity, we have devised the following four "storyotypes" (short for "story stereotypes".) These storyotypes are used to characterize each story and provide a means to split a "bloated story" into smaller but still valuable pieces. While the following storyotypes descriptions frequently refer to use cases, these storyotypes can be applied to any story whether they are more like a use case like or a larger XP story. Use cases just happen to be the best understood and most broadly used form of prose-based requirements capture so they form a good point of reference for these storyotypes descriptions.

Storyotype: New Functionality

This storyotype describes new functionality that is fairly independent of functionality previously described in other stories. In the use case world, these stories could be characterized as the happy path of one use case or several interrelated use cases. If several use cases, the use cases must be co-dependent (like chickens and eggs): it

would be difficult to test one without the other. A common example is the CRUDing (Create, Read, Update and Delete) of a business entity; it would be very difficult to update an entity that has not yet been created and it would be difficult to verify the update was successful without being able to read it. So, the create, update and read of a basic business entity might be grouped into a single "basic functionality" story.

The use case functionality included in this story should be restricted to a single scenario, with no conditional processing. The other storyotypes describe additional functionality related to (extensions of) this basic new functionality.

If a user interface is required as part of this story, the user interface should be "the simplest UI that could possibly work". That is, the most basic windows, fields, buttons, or menu items required to provide the functionality. Anything else related to the UI belongs in the *UI Enhancement* storyotype.

Storyotype: Variation of Existing Functionality

Stories with this storyotype describe a variation of functionality introduced in another story (most commonly, in a *New Functionality* story.) This can involve one or more extensions or exceptions (as described in [4]). This is the kind of story that introduces conditional logic into the software as each of these variations typically involves checking some condition and executing a different path when the condition is true.

When a *Variation* story involves several use cases, they will typically be the same use cases as described in the *New Functionality* story that the *Variation* story extends

User interface work related to this storyotype should be restricted to the addition of any data field to the screens required to enter or view data used to make the decisions.

Storyotype: New Business Rule

New Business Rule stories (often called "input validation" or "edit checks") extend *New Functionality* and *Variation* stories with additional constraints that need to be enforced by the software. This kind of story introduces conditional logic into the software in the form of guard clauses or assertions as each of these variations typically involves checking some condition and raising some sort of error condition when the condition is true. Any user interface work included in this storyotype should be restricted to whatever is needed to communicate the error condition to the user and the means for them to rectify the proble

Storyotype: User Interface Enhancement

User interface design and development is a complex discipline that can quickly become a major "time sink" if not managed well. It is one of the areas ripest for scope creep and the most fruitful for adjusting scope to match available resources. As such, it is very worthwhile explicitly separating the stories that relate to developing complex user interfaces from those that develop the underlying business functionality.

Stories with this storyotype should focus on a specific form of enhancement of the user interface and should not include any business functionality. If there are several "dimensions" of interface improvement required (e.g. drag&drop, multi-selection list boxes and voice recognition,) each should have a separate story or stories to enable the customer to chose the functionality they need most without dragging in other bits of less important (to them) functionality.

Refactoring Stories Based on Storyotypes

Having identified the storyotypes occurring in each story, we can make conscious decisions to split the stories into single storyotype stories or leave multiple storyotypes in some stories. There is a cost to having too many (and therefore too small) stories; combining them into larger stories results in fewer stories to estimate and keep track of.

We rarely find it useful to combine stories with different storyotypes. The main exception to this is when the single-storyotype stories are so small as to only require a single task to build them. This occurs most frequently during the bug-fixing or minor enhancements phase of a project.

We do find it useful to combine two stories with the same storyotype (e.g. two Business Rule stories) as it can be pretty arbitrary whether we call them a single story or several. Again, the size of the stories is a key determinant; we don't want the resulting story to be too large to be completed in a single iteration and we don't want to force the customer to "pay for" work they might not want just because it is lumped in with other functionality in the same story.

Managing User Interface Enhancement Stories

The style of the user interface is a "cross-cutting concern" that spans the different kinds of functionality provided by the system. Changes to the style of the user interface can involve visiting a lot of software. The key challenge when building ***User Interface Enhancement*** stories is to avoid excessive revisitation of each part of the user interface in successive attempts to build a highly usable user interface. It may take several (e.g. 3 or 4) tries to find a user interface metaphor that the users are happy with. Without careful management of the process, we may have to apply each ***User Interface Enhancement*** story to every part of the application's user interface as we learn what the customer really wants.

We have found the most effective strategy is to build the system with a simple UI initially and to do some UI enhancement stories targeted on a particular part of the system. This provides a way to get feedback on the UI technology and style without making a massive investment in the UI for the entire system. Once the users are happy with the UI in the pilot area of the system, the same UI paradigm can be applied to the rest of the system (typically in later iterations or releases). This can greatly reduce the churning of the UI code those results if the UI evolution involves the entire system.

(This is one area where it really is worthwhile avoiding rework by using Options Thinking [5] to delay the bulk of the work until the high impact decisions have been made.)

A Caveat on Combining Stories

Regardless of the storyotypes involved, we would only choose to merge two or more stories when they have identical business value and the level of specification certainty is the same. We also want to be sure that the value/certainty won't change before we build them. This is an excellent argument for "early splitting; late merging"!

Example

Consider an application that prepares invoices for various customers of a service. To show the applicability of storyotypes regardless of the approach used to come up with the stories, we will provide both a "use case" and a "bloated story" description of the functionality requested for the application. The intention is not that one would first generate the "bloated story" description from the use cases but rather that either could act as the starting point for the refactoring exercise.

Use Cases Example

The system includes a number of use cases including: Maintain Customer, Maintain Billing Cycle, Generate Invoices and Send Invoices. The Maintain XXX use cases include the ability to create, modify and either delete or obsolete the corresponding business concept as appropriate.

Use case "Generate Invoices" is used to produce the actual invoices that can then be viewed, regenerated, finalized and sent. Invoices may contain charges based on simple subscription (e.g. monthly charges), usage (e.g. so much per unit) and manual charges (special cases). It can be used to generate the invoices for all customers or only selected customers.

The user would like to be able to select the customer using a multi-selection list, by pressing a button to add the customer to the list of invoices to be generated or by dragging and dropping the customers onto the list of invoices to be generated. They would also like the system to remember the last group of customers used. And the system should not allow generating an invoice for a customer who has not yet been approved by the sales manager.

Use case "View Invoice" allows the user to see the list of available invoices and to select one for viewing in more detail.

Use case “Finalize Invoice” is used to “lock down” the invoice so that it cannot be regenerated. An invoice cannot be sent to the customer until it is finalized.

Bloated Stories Example

A team that is not familiar with use cases may have come up with the following stories for the same functionality.

Story 1: Invoice Generation: Generate an invoice consisting of a single subscription charge for one or all customer. View the resulting invoice. The user can select the customers whose invoices are to be generated using a multi-selection list box or using Add/Remove buttons to move the customers from the All Customers pane to the Selected Customers pane. The system should remember the last set of customers for whom an invoice was generated. An invoice cannot be generated for a customer until the sales manager has approved them. An invoice cannot be generated for a customer until all mandatory data elements have been provided. These include name, contact information (mailing address, phone #), title, and company name. Customers can be created with as little as just a name but they cannot be invoiced.

Story 2: Send invoice to customer: When the user is satisfied with the invoice for a customer, they may finalize it and then send it to the customer. Once finalized, the invoice cannot be regenerated or modified in any way.

Story 3: Usage-Based Charges: Generate an invoice that includes usage-based charges. The usage data is read in from a flat file and the usage rate can be set via a user interface. Generate the invoice and view it to verify the rate is being applied correctly. View the resulting invoice.

Characterizing the Bloated Stories using Storyotypes

Consider a story that describes the process of generating an invoice. This “use case story” includes many storyotypes:

Generating the invoice for all customers is an example of the *New Functionality* storyotype. Generating them for a subset of customers is an example of the *Variation of Functionality* storyotype.

Because there are three different UI metaphors being described, we can infer that there are at least two candidates for *UI Enhancement* stories.

Splitting the Story based on Storyotypes

Now that we’ve identified the various storyotypes, we can refactor the story into the following single-storyotype stories:

New Functionality: Generate a very simple invoice consisting of a single subscription

charge for the customer. View the resulting invoice. Note: This is an example of a "bootstrap story" as described in [3].

New Functionality: Finalize and Send an invoice to a customer.

Variation: Generate an invoice that includes usage-based charges. The usage data is read in from a flat file and the usage is charged at a rate of $1 per unit of usage. View the resulting invoice.

Variation: The usage rate can be set via a user interface. Generate the invoice and view it to verify the rate is being applied correctly.

Variation: Use a multi-selection list box of customers to select the customers whose invoices are to be generated.

Variation: Remember the last set of customers for whom an invoice was generated.

UI Enhancement: Select the customers for whom to generate the invoices (or finalize the invoice) using a simple dual list box with add/remove buttons UI metaphor.

Business Rule: An invoice cannot be sent to a customer until it has been finalized.

Business Rule: An invoice that has been finalized cannot be regenerated or modified in any way.

Business Rule: An invoice cannot be generated for a customer until the sales manager has approved them. This also requires a simple UI to approve the customer (probably described in the Maintain Customer use case.)

Business Rule: An invoice cannot be generated for a customer until all mandatory data elements have been provided. These include name, contact information (mailing address, phone #), title, and company name. Customers can be created with as little as just a name but they cannot be invoiced.

Business Rule: Only the sales manager can approve the customer. This implies some kind of login capability so that the system can be aware of who is using the system. Authentication (that is, security) could be another story.

Combining Stories Based on Storyotypes

Now, we can make conscious decisions to keep each instance of a storyotype in a separate story or to merge two (or more, but not recommended) storyotypes into a single story. In our example, we will choose to treat the two business rules related to when an invoice can be generated as a single story (that still has a single storyotype). We might call this story "Invoice Generation Business Rules".

We would only choose to merge them when we know their business value and certainty is the same and we are sure that they won't change. For example, we could choose to include both subscription charges and usage charges in the same invoice generation story. We would do this knowing the consequences of having done so

rather than out of ignorance.

Conclusions

The story is the foundation for describing, planning, and managing an XP project. Getting the granularity of the stories right is crucial for making the release planning game function efficiently. The four storyotypes we present here are a useful tool for understanding the size and complexity of the stories planning regardless of whether the stories are based on use cases or are bloated for other reasons. They give the neophyte XP team a set of heuristics they can use when making decisions about the how to refactor stories while doing release planning. These storyotypes came from our experiences using XP while building enterprise information systems; teams working in other problem domains may find it useful to identify storyotypes specific to their domain.

Acknowledgements

The author would like to thank all the ClearStream colleagues who shared their experiences and insights in managing stories on a variety of XP projects and especially Ted O'Grady who encouraged me and gave me valuable feedback on early drafts.

REFERENCES

1. Beck, Kent. Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000; ISBN 201-61641-6.
2. Beck, Kent. Martin Fowler, Planning Extreme Programming, Addison-Wesley, 2001; ISBN 0-201-71091-9.
3. Andrea, Jennitta. Managing the Bootstrap Story in an XP Project, in Proceedings of XP2001, 2001.
4. Cockburn, Alistair. Writing Effective Use Cases, Addison-Wesley, 2001; ISBN 0-201-70225-8.
5. Poppendieck, Mary and Tom. Lean Software Development, An Agile Toolkit, Addison-Wesley, 2003; ISBN 0-321-15078-3