Framework XP –Building Frameworks using XP

Gerard Meszaros
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 560-2408
gerard@clrstream.com

Jennitta Andrea
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 264-5840
jennitta@clrstream.com

Shaun Smith
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 264-5840
shaun@clrstream.com

ABSTRACT
This paper describes our experiences using eXtreme Programming (XP) to build frameworks and how we have had to modify XP to better suit this purpose. We call this variant of XP *Framework XP*. It builds on ideas first described in the XP books and augments them with concrete suggestions for dealing with the key issues such as how to derive the *framework stories* from the *user stories.*.

Keywords
eXtreme Programming, XP, Software Frameworks

1 INTRODUCTION

eXtreme Programming

Extreme Programming has its roots in Information Technology (IT) systems development where it was conceived as a way to make the IT organization ("development") much more responsive to the needs of the business (represented by the "customer"), by letting the customer select exactly those stories (features) that they need implemented for a particular release of the system. By making the stories small and concrete, an initial version of the software system can typically be built and deployed in just a few months. Additional functionality is then added in subsequent releases. The design evolves over multiple releases as more and more stories are built. Over time, frameworks may emerge as the common implementation of many related stories. This process is described in detail in the various books on XP, starting with [1].

Frameworks

A software framework is a mechanism used to significantly reduce the amount of software that needs to be developed when a system contains many examples of similar behavior. Frameworks [2] are typically characterized by:

1. Reusable, common behavior (typically domain-specific), into which one inserts
2. pluggable customized behavior, which is
3. called by the framework ("inversion of control")

Examples include testing frameworks such as JUnit or SUnit, IDE frameworks such as IBM's Eclipse or application frameworks such as IBM's San Francisco.

There are several key circumstances in which frameworks are crucial:

1) A *single system* requires many instances of similar functionality. A software framework makes building these variations more efficient.
2) A software company is building a collection of products for different markets or types of customers. The products share a set of common behaviors and each has it's own peculiarities. A software *product line framework* allows the members of the *product line[3]* (or family) to share a significant amount of software thus reducing development and maintenance costs as well are reducing the time it takes to bring a new product to market.
3) A software company recognizes a market exists for a *framework product* that will make it easier for it's clients to put functionality into their own systems.

Frameworks vs. XP

In "classic XP", frameworks are built by refactoring the software built in response to user stories with similar yet possibly conflicting requirements. The framework "emerges" as more user stories are built and the common software is factored out into reusable classes which call "plug-ins" that provide the different behavior. This approach depends on *collective ownership* wherein the clients of the emerging framework can be refactored simultaneously with the framework [1].

Traditional XP addresses *single system* frameworks quite effectively and will even work for *product line frameworks* as long as the product line is not so large that XP cannot be used to develop it.

But in cases where the product line is too large to be built by an XP team practicing *collective ownership* or when building a *framework product*, the "classic" user-story-driven approach to XP doesn't work because we can neither afford to wait for all the user stories nor can we refactor all the client code as we evolve the framework.

In these cases, we need a way to apply XP practices to defining and prioritizing the functionality of the framework distinct from the products or applications that will be built upon it. Yet the principles of YAGNI ("You aren't going to need it") and "build the simplest thing that could possibly work" would appear to be in direct conflict with what we need to achieve.

ClearStream Consulting Background

ClearStream Consulting has experience building a number of domain-specific frameworks that are in use by numerous customers[4]. We have recently applied XP techniques in the building of several frameworks. We have also been engaged in XP mentoring of clients that build frameworks. From these two perspectives we have discovered several ways we need to modify and enhance 'pure' XP.

2 FRAMEWORK CHALLENGES

Building a software framework for delivery to multiple customers, whether internal (*product line framework*) or external (*framework product*) is quite different from building a *single system* for a single customer. Amongst the differences are:

- Lack of a single customer
- Story concreteness vs. generality
- Difference in definition of "business value"
- Up-front vs. emergent design

Lack of a Single Customer

When the objective is to build a framework for sale or delivery to many other organizations, there will not be a single customer who can describe what the framework needs to do and decide on the prioritization of the capabilities. Often, the end customer isn't even willing to share details of what they want to build using the framework lest the information fall into the hands of their competitors. The requirements and prioritization coming from multiple customers often conflict and contradict each other.

Story Concreteness vs. Generality

One of the frustrations that have been commonly expressed by people trying to build frameworks using XP is the concrete and instance-based nature of user stories. Framework developers say things like "We are paid to generalize; that is the value we provide to the company!" The concreteness of user stories tends to get in the way. They are also too far removed from the capabilities to be delivered by the framework.

Difference in Definition of "Business Value"

The *framework product* supplier is evaluated on the ability of the customer to produce many different systems based on the framework. Being able to build a single concrete implementation often has no business value as it can only be sold to a single customer. Thus, the focus on delivering a simple and very concrete subset of the system functionality as early as possible (thus requiring very little analysis and design up front) is just not valuable. The framework provider typically wants to facilitate the ability to do several similar things with the framework to demonstrate flexibility right away. An example would be the need for a commerce framework to be able to accept both cash and credit card payments in the first release.

Explicit vs. Up-front Design

An XP project generally progresses one story at a time, with no explicit need to worry about the ordering of the stories. The system architecture and any underlying infrastructure evolve as more stories are implemented. The solution is generalized as time goes on through refactoring. This gradual evolution works well for one-off solutions. However, to build a successful software *framework product*, a bigger picture view is required up front in order to understand where the similarities are and where the points of customization are located (a,k.a *flexibility points or hot spots[6])* This is how the framework supports variability.

3 ADAPTING ING XP FOR FRAMEWORKS

XP needs to be adapted in several ways in order to successfully support building *framework products*. We call this variant of XP *Framework XP*. We need to recognize that stories occur at two levels: user stories and framework stories. As a result, the traditional XP role of customer needs to be split into two roles: end-customer and framework manager. Additionally, the planning game needs to be augmented with several new "moves" to reconcile the gap between user stories and frameworks stories.

Building software frameworks requires a great deal of abstraction and generalization. Many different usage scenarios need to be examined before the best design can be selected. The key to making XP work when building frameworks is to resolve the tension between the concreteness of the stories that describe examples of what the user wants to do and the level of generality required of a framework.

XP Roles for Framework Building

End-Customer

There are typically many end-customers of a framework product; these are the individual developers who will use the framework and the organizations they work for.

Framework Manager

For a *framework product*, someone within the organization building the framework needs to play the role of *customer* in the planning game[1,5]. Typically in the *framework product* case, the product manager plays this role, but it could also be a framework architect or an analyst who understands the clients' domain.

Development

Development may need to help the framework manager extract the framework stories from the user stories or to come up with additional user stories to demonstrate the desired flexibility..

Story Levels

When the objective is to build a framework for sale or delivery to another organization, the XP stories should fall into one of two levels:

1) Stories that describe a concrete instance of what the end-user would build using the framework. We call these *user stories*.

2) Stories that describe a single capability of the framework itself. We call these *framework stories*.

The problem is that the *framework manager* often cannot write the framework stories directly. The best they can do is to collect the concrete requirements (user stories) from the various end customers and feed them to development during the planning game. Imagine this as development responding to the customer by stapling a number of user stories together and saying, "We believe these are all the same story; we estimate the cost to be x."

Abstract End-User Story
Another way of describing flexibility is the "abstract story" in which the customer describes how something might vary. In classic XP, such abstract stories are discouraged; the customer is asked to write the specific stories that describe each way they want it to work. In framework XP, it would be up to the framework manager (possibly assisted by development) to fill in the specific stories.

Creating Framework Stories from User Stories
While the simple metaphor of stapling a number of user stories together into a single framework story is appealing, it is often not quite this straight-forward. More typically, a set of user stories will collectively imply a set of framework stories with each user story possibly requiring several framework stories to be realized. If the *framework manager* is not technical enough to be able to derive the framework stories herself, the development team may need to help her understand the derivation from the user stories so that the *framework manager* can choose the sequence of framework stories that provide the most business value.

Testing Framework Stories
A framework requires two levels of acceptance tests: those that verify the framework stories and those that verify one or more concrete user stories. In classic XP, the latter would typically be written by the customer and executed while building the application that is based on the *framework product*. But it is better if the development team writes a very small sample application, which exercises the framework and allows direct testing of the user stories that demonstrate the flexibility supported by the framework.

4 MODIFIED PLANNING GAME
In effect, we add several new moves to the *exploration phase* of the planning game in addition to the traditional ones of "write a story" and "split a story" (the customer) and "estimate a story" (development) and *prioritize stories* (customer). The new moves are:

- Analyze variability (to extract implied framework capabilities)
- Define framework stories

These two moves create the framework stories that are used in the release-planning phase of the planning game.

Analyze Variability
In this move, development analyses each concrete user story and extracts a set of concrete framework capabilities that would be needed by the customer to achieve their goal. The key is to have the *framework manager* put enough user stories in front of the developers so that they can generalize them into framework stories, which they then estimate. The *framework manager* can then decide which framework release to place the framework story into. The first step is to partition the story into those things that could possibly be provided by the framework, and those that must be supplied by the customer. The second step is to define the capabilities that the framework provides. At this point, these can be thought of as a "wish list" such as: "It would be great if the framework could automatically determine when to invoke the user logic."

One technique we have found useful is to sift through the user stories looking for candidate topics that a framework might provide help with and cross-reference the user stories with the topics. This topic list then becomes the starting point for discovering the framework stories. When we find places where otherwise similar user stories conflict, we have identified candidate "hot spots" or "flexibility points" needed in our framework.

Define Framework Stories
In this move, development collects the extracted concrete framework capabilities ("wish list") and defines the simplest set of framework stories that would provide the required capabilities. This step requires enough user stories to imply the nature of the framework services that will be required. In this case, "the simplest thing that could possibly work" is more complex than for a single system reflecting the very nature of a framework.

This move corresponds roughly to the analysis phase of traditional software engineering processes and results in what could be called the use cases of the framework. These often correspond to the "change cases" of the applications that would be built using the *framework product*. While defining the framework stories may require defining the API of the framework, one need not design the framework implementation yet.

Framework Release Planning
To remain faithful to the XP principles, it is important for the *framework manager* (the "surrogate customer") to retain control of the development sequence and priorities by choosing the framework stories for development in each release of the framework. Since the end customer is not involved in the planning game, it is up to the *framework manager* to make the difficult tradeoffs between the possibly conflicting priorities of the end customers and present a single voice to development.

5 EXAMPLE
In this example, we provide a set of user stories for a gas pipeline billing system and the framework stories for a billing framework that were derived from them.

Abstract End-User Story
The following abstract story might expand into the user stories that follow.

Flexible Rate Calculation
I want to be able to change the way the rates are calculated as my business changes.

User Stories
Flat Rate Calculation, Using Volume
The charge against the customer's account is calculated by multiplying the number of million cubic feet of gas moved on the pipeline regardless of where it was received and where it was delivered.

Fixed Monthly Charge, Using Volume
The customer may be charged a flat monthly fee for the contracted capacity whether or not they use it. These are in addition to any usage based charges (which are typically lower than for contracts that don't have any capacity charge.)

Contract Based Rate Calculation, Using Volume
The charge calculated for the volume of gas moved on the pipeline is computed using the rate specified in the customer contract associated with the point at which the gas was received.

Distance Based Rate Calculation, Using Energy
The charge is calculated by multiply the energy content (in Kilo Joules) of the gas moved on the pipeline by the published distance between the receipt point and the delivery point.

Variability Analysis
These user stories have several key points of conflict. Assuming they are all true statements of requirements, these points of conflict become our points of variability.

- Several stories talk about the units for usage upon which the charges are based (volume vs. energy content of the gas)
- Several stories talk about different usage multipliers (single rate, receipt point, contract specific, distance of haul)
- One story implies a flat monthly fee regardless of usage while the remainder only talk about usage

These lead us to define a framework story for each of these variability points.

Framework Stories
The following framework stories describe the specific dimensions of flexibility seen in the preceding user stories.

Flexible Charge Calculation Algorithm
The charge calculation has a fixed monthly component and a usage based component.

Flexible Usage Charge Calculation
The usage-based charge may be calculated using a single rate, a rate determined based on: the receipt point, the customer's specific contract, or the published distance between the receipt and delivery points.

Usage Charges Based on Energy or Volume
The amount of the usage charge may be based on either the volume of gas moved or the energy content of the gas moved.

6 CONCLUSIONS
XP can indeed be used to develop frameworks for delivery to multiple, possibly anonymous, customers but it must be modified to be effective. It is important to distinguish between the user stories that describe concrete situations and the framework stories that describe the framework capabilities (that in turn support the general cases of the user stories.) This requires additional moves in the planning game for turning user stories into framework stories and a *framework manager* role to act as the single customer who prioritizes the functionality to be developed.

ACKNOWLEDGEMENTS
The authors would like to thank the clients who's projects gave us the opportunities to gain the experiences described here as well as the colleagues who acted as guinea pigs while we tried out these ideas.

REFERENCES

1. Beck, Kent *Extreme Programming Explained - Embrace Change.* Addison Wesley 1999.
2. Roberts, D. and Johnson, R. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks* in "Pattern Languages of Program Design 3", Addison-Wesley, 1997
3. Northrop, L *What is a Software Product Line?* http://www.sei.cmu.edu/plp/frame_report/what.is.a.PL.htm
4. Meszaros, Gerard et al "*Business Object Framework as an Enabler of Business Application Development*", OOSPLA'98
5. Beck, Kent, and Martin Fowler *Planning XP* Addison Wesley
6. Pree, W Design Patterns for Object Oriented Software Development, Addison Wesley, 1994