Experiences with Software Product Line Development in Risk Management Software

Gerard Quilty
ORisk Consulting
Dublin, Ireland
Gerard.Quilty@gmail.com

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland.
mel.ocinneide@ucd.ie

***Abstract*— Software Product Lines are intended to reduce time to market, improve quality and decrease costs. In this paper we examine the evolution of a single system to a Software Product Line, and evaluate if these benefits have occurred in this case. We describe in detail how this evolution took place and relate our experiences to those described in the current literature. Three tenets used by the company involved helped avoid some of the known pitfalls. A configurable core asset version of functionality is compared to the previous customizable version of the same functionality. From analyzing empirical data collected over a ten-year period, we find that efficiency and quality have improved, while costs have been reduced. The high initial investment associated with evolving to an SPL has been postponed by taking small steps towards an SPL architecture. In addition, this approach has enabled us to expand our product into a wider market and deal with more complex customer requirements without incurring a corresponding increase in staffing and costs.**

Keywords: operational risk management, software product lines, industrial experience.

I. INTRODUCTION

ORisk Consulting is an independent business unit of a large multinational company that provides Operational Risk Management Software to financial intuitions. The product, Blade, has undergone numerous changes in its ten year history. It was originally built as a bespoke project for a single customer and now is a recognized leader in operational risk management software. The processes and coding standards involved in developing and implementing Blade resemble those involved in developing a family of software products or a software product line.

A Software Product Line (SPL) is a set of software intensive systems that satisfy the specific needs of a particular market segment developed from a common set of core assets in a prescribed way [1]. This paper maps Blade to this definition of software product line and demonstrates some of the concepts applied in order to limit the initial development cost associated with software product lines [2]. It also highlights our successes and failures in taking this approach, in particular with respect to realizing the proposed benefits of software product line development.

The culture that led to the development of Blade as a software product line grew within the company without reference to academic literature. This mirrors the first SPLs identified by Clements [1], which grew out of industrial need rather than academic endeavor. We discuss the benefits of SPLs and illustrate that those benefits are achievable for a small development team. Development time data and bug maintenance data gleaned from the lifetime of Blade are used to investigate this.

In order to confirm or deny the benefits of SPL concepts in Blade development a comparison was carried out. As Blade is a mature product it has had sections of its code rewritten over time. Originally it was developed in an ad-hoc fashion with the customer being the key driving force. In this paper such an approach is described as a *customization approach*. One of the key identifiers of such an approach is that customer changes require of custom code written to be for them, and that variation is dealt with by selection statements checking which customer is currently using the system. Gradually the focus of development changed to a more configurable framework that is comparable to SPL development. This is where configurable core assets are developed that have the potential to deal with various customer requirements so that change requests can be dealt with by configuring the related core asset instead of rewriting it. In this paper we compare the customization approach and the configurable approach for one section of the Blade product that has been subjected to both approaches. This section, risk and control scoring, has experienced considerable variation between customers, due to the dynamic nature of risk management.

This paper is structured as follows. Section II describes the terminology involved in operational risk management, and provides an overview of the software product, Blade. Section III describes the processes and policies that have been adopted to manage core asset development and eliminate some of the recognized problems associated with SPLs. Section IV presents a detailed comparison of the two different approaches to developing code. Both these approaches are compared in terms of initial development cost, maintenance cost and other noticed effects of the changes. Finally, in section V, we present our conclusions and discuss possible further research and opportunities to improve efficiency by reference to the SPL literature.

II. BACKGROUND

In this section we provide background information related to operational risk management and to the product that is the subject of this paper. Section II.A introduces risk management concepts and definitions. It also covers what is involved in managing risk in financial companies. Section II.B deals with how the product helps risk managers manage their risk as well as background on the forces that caused it to evolve to a Software Product Line.

A. Operation Risk Management

Operational risk is the risk of loss resulting from inadequate or unsuccessful internal processes, people or systems or from external events [3]. For example, an earthquake is a *risk* for offices in certain countries. A *risk control* is a protective measure for reducing risks to, or maintaining risks within, specified levels. Earthquake insurance is a control on a risk that mitigates the impact of an earthquake if one occurs. An occurrence of an earthquake is called a *risk event*. In a risk event, a risk is transformed into consequences that may relate to internal or external sources.

All companies are susceptible to operational risk events and failed processes. For example, Toyota, one of the world's largest car manufacturers, was obliged to recall over 2.3 million US cars due to a failed internal quality process. Toyota lost 21% of its share price and suffered immense damage to its reputation from this event. It could have been avoided if the risk had been identified and proper controls put in place to prevent it [3].

Financial institutions have a regulatory requirement to manage risk within their organization. This requirement covers both internal and external risk. Banking institutions are governed by the Basel II accords [4], while insurers are governed by Solvency II [5]. The Advanced Measurement Approach or AMA is the highest level of operational risk management according to Basel II. It states that the institution has to convince its supervisor or regulator that it is aware of and is managing its risks. To comply with AMA, the company must at a minimum have the board of directors and senior management involved in the process. An operation risk system that is conceptually sound and that is implemented with integrity is required, and the company's risk team must have sufficient resources and training [4].

Early identification of risks and the implementation of controls that lessen the effects of those risks are vital in preventing damaging risk events. Identifying risks is a complicated activity for businesses. It requires a large amount of information from different sources, both internal and external to the company. External events are analyzed to determine if the underlying risks that caused the event could exist within the company, and if such an event is likely to occur for that company. Companies also record and maintain other information such as indicators and scenarios that help them to identify risk and controls and quantify the impact and likelihood of risk events

An *indicator* is a measure of certain activity within or outside of a company over time. It is used to evaluate how likely a risk is to occur. For example, if an indicator based on unemployment figures goes up it is more likely people will not be able to repay mortgages. The probability of risks related to unpaid mortgages therefore goes up.

A *scenario* is a collection of related risks that have a very low possibility of occurring together but can have a devastating knock-on effect when they do. Such unpredictable events are sometimes referred to as black swans as they can occur because institutions make assumptions about their environments (all swans are white) that turn out to be incorrect [6].

Companies will usually maintain a master register of all risks that they are aware of. These register risks would be related to their business area and the wider industry in which they operate. Libraries of controls and tests for those controls are also managed. Local risk managers select from these libraries of companywide risks to create local risk maps for individual departments.

To reassure regulators and supervisors that they are managing risk appropriately, companies perform risk assessments periodically. A risk assessment can be control-centric or risk-centric depending on the relative importance given to the controls and risks within a company. In a risk assessment controls and risks are scored using what can be a very complex scoring model. These complex scoring models serve a dual mandate. Firstly they are an attempt to quantify what can be a quite complex problem, such as the probability of an earthquake occurring. They are also designed to convince regulators that the organization knows what its risks are and has adequate controls in place to deal with them.

Risk Management is not about avoiding all risk events as such a task would be impossible. It is about setting a threshold of acceptable risk and limiting the impact of risk events when they do occur. A risk manager, after completing their risk assessments and having signed off on the level of risk will report that level of risk up to senior management. Often the risk assessments are rolled together to give a broader sense of the risk level in the company. This conforms to the first requirement of AMA to ensure that senior managers are informed and aware of the level of risk.

Control-centric assessment deals with users signing off on controls and confirming that the control has been performed. A risk with unperformed controls can have a much higher impact than when the controls are performed. This means that senior management can be signing off on a lower level of risk within their company than actually exists. Such action may result in formal regulatory action and have severe consequences for the company.

B. Blade

Blade is the risk management software application that is the subject of this paper. It is a web application that supports many different customer frontends. Web applications have previously been built as SPLs, for example the vacation home rental application, HomeAway [7]. In the case of HomeAway, the driver for development was the acquisition of different companies with web sites in the same business context. In our case, the driver for developing configurable core assets was the need to support varying customer requirements without an increase in costs.

Blade is targeted at financial institutions such as banks, insurers and asset managers. An operational risk management system needs to conform to the requirements for AMA and help companies identify, categorize and assess their risks. The software provides support for all the varied operational risk management processes. It is a web based application based on the .NET framework and Microsoft SQL Server. It was first developed in 2000 using ASP as the base language. Since then it has been fully redeveloped in .NET with C# being used for backend auto generated data classes and VB.NET used as the code behind the web pages. In 2008, the Model View Presenter (MVP) pattern was implemented as the architectural pattern for new pages.

Blade is split into four solutions. One contains core components such as authorization and reporting features that the other three use. Modules for the different processes in operational risk management, such as risk assessment, control signoff and managing risk events, are contained in the main solutions. The two other solutions are specialized modules for the recording of indicators and scenarios. These tasks are outside of the usual remit of an operational risk management team but can increase the accuracy of risk calculation.

The development of indicators and scenarios led to the refactoring of many core assets out of the main solution into a core asset solution. These assets needed to be available to other solutions. Some core assets were developed specifically for one of the modules to solve a specific requirement, e.g. a user-definable list and a lazy loading tree. These core assets have been introduced back into the main solution. The user definable list has been so successful that it now has over eighty implementations and is the default solution for any new lists that are needed in the system.

In terms of size, the total code base is over two million lines of code, with one thousand tables and two hundred stored procedures. This is a large project for a small development team to maintain and upgrade. As a mature product, modules and concepts have changed over time. These changes range from maintenance fixes to complete rewrites to take advantage of developing technology. This leaves areas that have in the past needed to be customized to individual customers but now are configurable instead.

Scoring is one of these areas, and is the focus of the comparison in this paper. It is an area that experiences a huge amount of variation between customers. A risk assessment is where a user will score their risks and controls on a regular basis. The base unit of a score is a single input called a *scoring measure*. This is where the user records a single fact about that risk or control i.e. how damaging the risk is, or the probability of the risk occurring. A user-defined calculation uses these inputs to calculate a *score value*. This score value is then compared against a set of thresholds to evaluate a textual description for that score. These calculations are recursive; one score value can be an input into a further calculation to create a new score value.

Inputs to a scoring measure do not necessarily come from the user entering a value on the front end. They can come from thresholds set against the organization that the risk is part of, from external or internal events targeted against that risk, from indicators that the customer records and from other external systems. The calculations for a risk score often depends on a special set of inputs called combined control scores. These are calculations using the set of all controls acting on that risk. This makes scoring a complex and challenging problem.

It was initially solved for customers by embedding their scoring models and their individual calculations into stored procedures and the code. However, in the first quarter of 2009, the underlying framework of scoring was changed to be user configurable. In order to reduce the development risk it was first changed for risks and then after the concept was validated it was applied to controls.

Many success stories exist that testify to the benefits of SPL development and the introduction of SPL methodologies [1], [7], [8], [11]. These stories highlight the benefits in terms of costs efficiencies, decreases in development time and quicker time to market. However they also identify a high initial development cost for the development of SPLs. Small companies do not necessarily have the capital for such development and hence techniques or processes that avoid the initial high cost are required in order for small companies to move in a SPL direction and realize these benefits.

The refactoring of customizable sections such as scoring into configurable frameworks have helped us realize the benefits of SPLs. Different sections were refactored slowly over time in order to spread out the initial development cost. This also lead to a spreading out of the benefits, however it was unavoidable as the initial investment required to implement SPLs fully in a short space of time could not be supported by revenue streams.

III. Applying Software Product Line Development to Blade

The original Blade system was as a standalone project for a single customer. Three years after its initial development it was adapted for release as a software product. The match between Blade and what the market required was not perfect. It shared many of the core concepts of operational risk management but each customer had a different understanding of the details. Within the company Blade is still thought of as a single product instead of as a product line, despite the fact that allegories of the organization and processes used can be found throughout the SPL literature and are highlighted in the following subsections.

Blade can be considered as an SPL due to the amount of variation between customers and the implementation process applied for each new customer. Every new implementation requires matching the capabilities of the system to the needs of the customer. Those needs are so varied that each new customer is viewed as a new product in the greater product line of existing customers. An implementation requires picking and mixing the existing capabilities into a new configuration for the new customer. Doing this well requires significant effort. In order to limit the cost of new implementations ORisk Consulting has geared itself towards utilizing the benefits of SPL development.

This section covers the details of the organizational approach applied and the processes that it covers. It focuses on

the specification and development of core assets and how the focus of development is kept on core asset development instead of on customization of code for individual customers.

The remainder of this section is organized as follows. In section A the development organization is discussed, while in section B other processes from the broader company perspective are described. The broader methodology has grown out of the success the development team have had with core assets.

A. Core Asset Development and Processes

Creating the core assets is a key activity of SPL development. It is from these core assets that new products are developed. The development team is small and does not include a dedicated core assets team. Responsibilities for core asset development must be shared amongst the team members on the basis of who is available to work on them.

Kruger classified three models for developing core assets, *proactive*, *reactive* and *extractive* [12]. In a proactive model, core assets are mapped out exactly before commencing their development. It is reminiscent of the waterfall life cycle and is geared towards having a dedicated core development team and hence ill-suited to small teams. In a reactive approach the core assets are built when there is a need for them, hence reflecting agile methodologies. The extractive process takes an existing product as the base for reuse. Parts of the product are extracted as core assets and then reused within that product or in new products.

Blade development uses a primarily a reactive strategy combined with some extractive processes. A reusable asset is identified by finding a capability that is required in two or more places. It is often found by developers recognizing that a required piece of functionality already exists, and deciding to combine both implementations into one. This involves reacting to the new requirement and extracting existing attempts at that requirement into a new core asset. Such a combination of reactive and extractive is preferable to proactive strategies for small development businesses as it limits risk and initial investment [11].

Staples and Hill report similar experiences with SPL development in a small company that does not have a defined SPL architecture [8]. In Blade also, the architecture has evolved based on the needs of the customers. This customer-focused method of dealing with variation has much in common with customer-centric, one-of-a-kind development in manufacturing [9], [10]. In one-of-a-kind manufacturing customers order a product made up of available parts. This matches how implementations of Blade are done, as modules are selected by customers and configured to match their needs.

Having just a single development team acts as a business unit as described by Bosch [13]. Such a setup has drawbacks as it does not focus on shared assets and the cost around making decisions involving shared assets is high [14]. Approaches to counter these drawbacks have previously been proposed for example using a “champion team” in place of a core team [15]. However we have not encountered these drawbacks. Over time three main tenets have been developed to guide our attempts at core asset development and to ensure that the benefits of core asset development are known and utilized throughout the company. It is possible that the tenets guard against these disadvantages. These tenets are:

- Avoid development risk, i.e. ensure that the development of the core asset does not impinge on our ability to deliver agreed changes.
- Create core assets that fulfill a need within the system and are useful to the development team.
- Communicate changes and new core assets across all teams and rotate the implementations of new instances of the core assets between developers

For the first tenet new core assets must be prototyped in a limited area first and then validated by the quality assurance department before being propagated to other parts of the system. An example of this tenet is how we created the *Generic List*. It was one of the first core assets developed and is a user configurable list control where a user can set what columns are displayed, how the list sorts and what search criteria are applied to it. The core asset makes it easier and quicker to implement a rich list user interface. It was first implemented during the development of the Indicators module.

The Indicators module allows users to define and record different risk indicators to try to gain better insight into the probability and impact of risks. It was developed as a green-field solution and as such allowed for the growth of new development concepts such as the Generic List core asset. After Indicators was released and the basic concepts of the Generic List were validated it was added to the main Blade modules. This recursive rollout allowed the Generic List core asset to be refined and to ensure that at each step there was the least amount of impact on the rest of the code base. Other companies have also taken a phased approach to the extraction of core assets for SPL engineering in other to limit risk and initial development cost [16], [17].

Limiting the development risk also limits both the cost to make decisions and the cost to implement those decisions. If the rule of thumb is to always make the decision with the least amount of impact on the system, then making that decision becomes easier. In some cases a decision will be taken that is not the one with the least associated risk. In this case it will lead to greater rewards with an acceptable level of risk.

The second tenet means that core asset must make a developer’s job easier by removing some time-consuming repetitive development task. For example, a standard .NET grid requires the setting up of template columns, bound columns, look and feel standards, sorting and a multitude of other minor tweaks. With the Generic List core asset, a developer essentially obtains all these tweaks for free. When the Generic List was applied to the Blade solution it removed over four hundred lines of code per page and replaced it with five lines of code and around twenty lines of metadata. This makes creating list pages much easier and quicker when using the core asset.

Developer-friendly core assets hugely improves developer uptake of those core assets. It also removes the barrier to acceptance for new core assets among developers. There can be a tendency to resist change and new ways of writing code;

however ensuring ease of use of core assets provides an incentive to developers to use the core asset and to identify potential new core assets within the system. This keeps the focus on core asset development within the team.

Without knowledge of a core asset, it cannot be used and the benefits of developing it are lost. This makes the third tenet the most important and, as such, we have processes in place to ensure its survival. After completion of the initial prototype which is usually the responsibility of one developer, that developer must inform the rest of the development team about that asset. This is done at the weekly developer update meeting. The communication also goes from the development team out to the business analysts and to the quality assurance team. The outwards communication means that the core asset can be used as part of the language between the different teams. As the core asset is part of the language set, business analysts will tell developers to implement an instance of a core asset or to add a new capability to an existing one. This knowledge sharing is how we ensure that the most reuse benefit is gained from each core asset.

Krueger stated that a key to combating entropy and developing core assets is to give someone the responsibility to identify core asset opportunities within the development team [18]. Developing successful core assets such as the Generic List that benefit development as well as the product, changes the company culture to be more focused on core asset development, as well as delivering on Krueger's point. As we limit development risk with each step that we make to create a core asset, the core assets that are developed tend to be very successful and deliver real efficiency gains. However this approach does lead to a longer development time and cost for the final core asset.

Handling variation is another key aspect of core asset development. Anastasopoulos defined four different binding times for variability in SPLs [27]. *Compile-time* binding is where the variability is defined before or during the compilation of the code. *Link-time* binding is where different libraries are used and linked into the code. *Runtime* binding is where the variability is resolved during the execution of the program. *Update-time* binding is where variable functionality is added when the software is updated.

Blade provides variation at runtime as well as using a variant of compile-time binding that we call *design-time* binding. Design-time binding is when the variation between different instances of a core asset is designed into the system. This design variation involved is usually a fundamental variation such as the difference between scoring a risk and scoring a control. A particular instance of a scoring asset needs to know what type of scoring asset it is. Design-time binding is different to the original definition of compile-time binding as it does not allow for the addition or removal of code from the system, but involves the hardcoding of the fundamental features of a core asset.

As a company we do not believe in limiting the capabilities of the product in such a way to make it hard for customers to change their minds about the capabilities that they wish to have. For this reason, all changes within a core asset, other than those described as fundamental to that instance of the core asset, are configurable at runtime. This means that we do not use link-time variability or compile-time variability based on preprocessor directives, as using those mechanisms would require us to release a new version of the product to the customer with the relevant Dynamic Link Libraries or preprocessor arguments.

Anastasopoulos also described a set of mechanisms for code level variation [27]. Blade uses *Parameterization* -- core assets are parameterized and different capabilities are made available depending on the values passed. Such a method allows for easy metadata configuration as the database can contain the necessary parameters to create an instance of a core asset, and all that is required at design-time is a reference to the particular set of parameters for a given core asset.

More complex core assets, like the scoring model, require inheritance and interfaces to implement variation. Risk scoring and control scoring are saved to different tables within the system. Therefore they are passed an instance of an object that implements a common interface and that point to their respective database tables. This object is used by the core asset to populate its base state as well as to save any changes to the scoring. Such a variation mechanism also uses design time variability as it requires an instance of the core asset to be passed a particular object based on what that instance is intended to be do.

B. Other Processes influenced by Software Product Lines

As part of the process, business analysts are kept informed of developed core assets. While the development and identification of core assets is a development task, once they are created the business analysts can make use of them. A hybrid agile/waterfall approach is used for historic and commercial reasons. Much of a quarter's work comes from customers. These changes need to be signed off by the customer before development starts. This requires a specification to be written for that change. Once the specification is signed off it goes to the development team. This is where the process becomes more agile.

In order to keep the development focus on core-asset development instead of single customer development the specifications are written in such a way as to provide the most benefit to the most customers. Customers face similar problems and hence requirements are often shared. It is the task of the business analyst to take a single customer's requirement and translate it into both the system vocabulary and the vocabulary of other customers. If the business analyst cannot determine if a change is right for different customers then a working group of customers is brought in and asked if their companies share the requirement and how they would like to see the change implemented. Where a change request is specific to only one customer and no other customer would ever what to do it then we will usually refuse to do the change. This is because over the lifetime of Blade we have learned that the development fee that can be charged does not cover the cost and effort involved in maintaining code for a single customer.

Writing specifications is a time-consuming task, and, in the past, there bottleneck problems have occurred. Developers can be left idle waiting on complex specifications to be finalized

and signed off. Core asset development helped us overcome this bottleneck by speeding up the specification writing process. For example, if a change requires a new list then a business analyst can simply use a Generic List in the specification and then the developer or quality assurance expert reading the specification knows that this is a list that requires user selectable fields, sorting and filtering functionality.

Like design patterns [19], using core assets has provided a common lexicon to describe complex pieces of code easily and efficiently [19], [20]. Unlike design patterns this lexicon is common across development, business analysts, quality assurance and even customers, thereby easing communication on all fronts. This does provide a barrier to new employees as they must understand the same concepts that the rest of the team works in. Documentation helps with this. An internal Wiki of all core assets within Blade is maintained and updated. It includes implementation and user guides, feature explanations as well as lists of all implemented instances of the core asset and the features enabled for each one.

Having customers and business analysts involved with core assets has led to a lot of focus on improving those core assets. For example, the Generic List has had numerous feature upgrades like the adding of saved personal filters, quick search and export options. All of these features are made optional if possible. If a feature should not be available for a particular instance of a core asset due to a logical rule then that feature is turned off in the metadata. For example a quick search should not be available for lists that are themselves the result of a different search. Keeping track of logical rules like this enables us to quickly determine what features a new instance of a core asset should have. This in turns limits the cost of implementing new core assets and of making decisions about those assets.

Making the features of core assets switchable and configurable by customers removes much of the burden of variability management [21], [22]. Blade is a web application and does not have limitations on the size of its installation footprint. This means that all customer variations can be shipped to all customers without the need to exclude branches of execution using techniques such as conditional compilation [23]. This allows customers to take advantage of not only their own knowledge base of best operational risk management techniques, but also the knowledge base of other Blade customers.

Unfortunately the negative side of making all the code available is the impact on the quality assurance team. This team requires a lot of time to test any change to a widespread core-asset. This is because any change to a core asset needs to be checked in all instances of that core asset against all known customer configured variations of that core asset. However, since the system is configurable at runtime, a customer might have changed a setting and this setting change may not be captured and tested by the quality assurance department. There is a risk, due to the complexity of this approach, that serious bugs will pass unnoticed through testing and have a negative reputational impact on the company. While this has proved not to be a problem in practice, it remains an area of testing where we still are seeking a good solution.

IV. Comparison of Development and Maintainence Effort Between Configurable and Customisable Approaches

In the first quarter of 2009 the risk assessment module of Blade was rewritten. A key part of this rewrite was to develop a new configurable scoring framework that could deal with all of the complexities of existing customer scoring as well as future customers' scoring requirements. This new configurable framework replaced the original scoring models in Blade. Eight different scoring models were developed for customers between the first version of Blade in 2000 and when it was replaced in 2009. These scoring models were migrated to the new framework.

The comparison presented here covers a time period of ten years from September 2000 to September 2010. The analyzed data covers over 21,000 bugs and 1,000 change requests. All bugs relevant to either version of scoring were extracted from this data. 1,009 relevant bugs were identified.

This section compares the two methods of dealing with customer variation, customization and configuration. It is performed across three levels. The remainder of this section is as follows. First, in section A the initial time to develop is examined. Next, in section B the bug maintenance effort for each version is calculated. The third section C deals with the other effects of moving from a customizable scoring approach to a configurable scoring approach.

A. Initial Development Time

The definition of initial development time used here for comparison is the time recorded against a change request from the start of implementing that change to the time it is first delivered to the customer as completed. It does not include the time taken to specify or negotiate the change request. The initial development for the customizable scoring models covers the development time for implementing eight different scoring models. The initial development time for the configurable approach covers development of the framework as well as the implementation of six new scoring models in that framework. There is also a significant development cost to migrate the eight original scoring models and existing customers onto the new configurable framework.

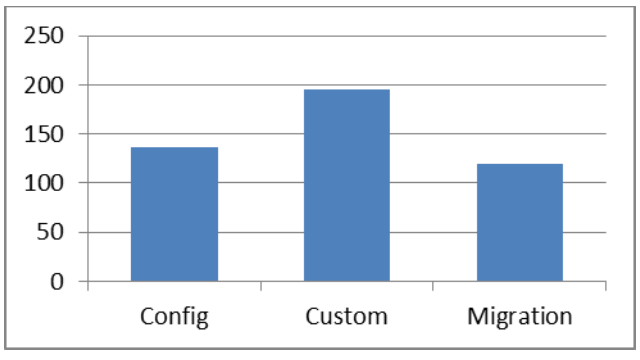

Figure 1. Initial development time (in hours) for the configurable framework, the custom implementations, and migration between the two.

Replacing the existing customization approach with a configurable approach required a considerable upfront investment as seen in Figure 1. This investment represents the combination of the migration and configuration columns and is greater than the cost of developing the original scoring models. The average time to create a new scoring model for the customization approach is 24 hours. For the configurable section it is 23 hours (including migration time). This increase in efficiency is too small to justify the amount of investment required.

Developing the configurable framework was a one-off cost of around 115 hours. This cost does not need to be reapplied for each new configurable scoring methodology. The average time to implement a new scoring methodology using the configurable framework when the development of the framework is excluded is around four hours. This is a six times increase in efficiency. Adding new scoring models is an increasingly common task. In the two year period since the creation of the configuration framework, six new scoring methodologies have been created. This compares to the eight that were built during the nine years before that. Thus the effort involved in developing a configurable approach has already paid off.

B. Maintenance Time

The level of maintenance required by a change to a product is sometimes overlooked. Part of the development process requires developers and quality assurance personnel to record time directly against the defects or change requests that they are working on. By recording time in this way we are better able to estimate workloads and plan quarters.

This data can be used to analyze the maintenance record of any module or component of Blade. The overall count of bugs related to an area and the time recorded against them provides a clear image of the maintenance cost of a section of code as well as any problem areas.

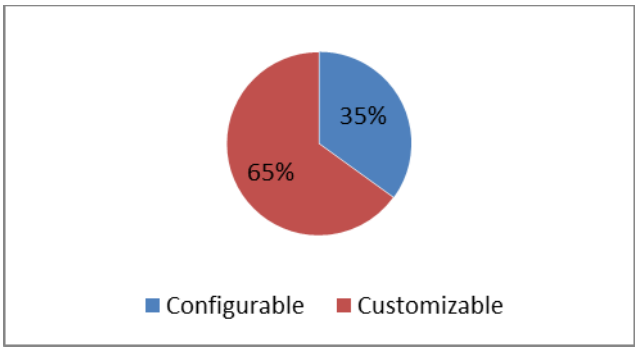

Figure 2. Comparison of maintenance effort required by configurable and customizable approaches.

Figure 2 highlights the comparable bug counts between custom scoring and configurable scoring. The configurable scoring count includes bugs raised during the migration of the old scores to the new framework for existing customers. The overall bug count for configurable scoring is significantly smaller than the bug count for the customized code. We can suggest two main reasons for this.

Firstly the custom implementation has a larger code footprint than the configurable implementation. The customizable approach deals with variation by using selection statements. Eight different scoring methodologies require eight different conditions. The result of which was a lot of confusion and a lack of readability in the code. Part of the development goals for configurable scoring was to eliminate this confusion by centralizing the code and removing some of the recurring bugs that continually customizing the code was throwing up.

The difference in the relative bug counts per priority in Figure 3 highlights this difference in code footprints. Considerably more trivial and critical bugs were found during the lifetime of the custom code than in the configurable code. These bugs were grouped around several problems areas. The trivial bugs primarily occurred as look and feel bugs. The select scoring measure values page. For one scoring methodology the page got duplicated requiring the same bug fixes to be done across two different pages. Each time a new set of scoring measures were added to the page it would break the look and feel for the other scoring methodologies.

The majority of critical bugs raised against the custom implementation were grouped around problems in the different calculations. Primarily those calculations were not returning the expected value. The root cause often came down to the function doing the calculation getting incorrect values for the measures passed to it, or getting the values in the wrong order.

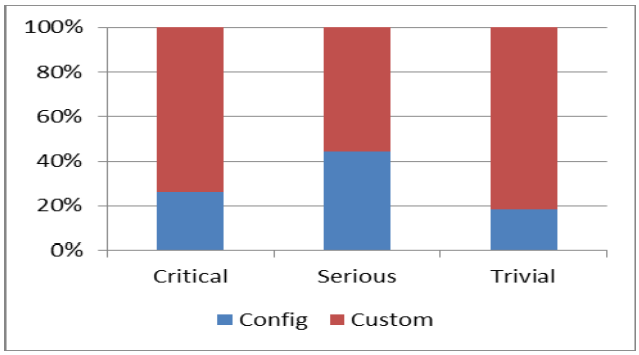

Figure 3. Breakdown of bug counts across bug priority.

These problems are avoided in configurable scoring as the user interface is dynamically built based on configured metadata. The metadata tells the software what measures to display, the valid values for those measures and in what order to display them. How the controls are added to the page is constant so look and feel cannot be broken for old scoring measures when a new scoring methodology is added. It also ensures that the correct measures are shown for the correct methodologies. The calculation itself is also configured in metadata. This limits the impact of defects in the calculation as it can be changed without having to change code. This dynamic adding of controls forms the basis of the configurable approach taken with scoring.

The configuration of the metadata that the scoring uses is an area unique to the configurable approach. It is here that most of the bugs with configurable scoring occur. Metadata bugs tend to be raised as serious and this is why the number of serious bugs is so high compared to the critical and trivial bugs.

All the scoring measures and calculations are built up using metadata that is entered by a business analyst or support engineer. It is a complex task that is prone to error. It is hoped that tool support will decrease the number of bugs raised due to metadata issues and improve the current average time to set up a new scoring methodology as well as decrease the maintenance cost.

Another possible reason for the large disparity in bug numbers is due to the fact that the custom implementation has had more time to mature than the configurable scoring implementation. Figure 4 highlights the difference in ages. It shows the estimated hours for both configurable and customizable scoring across the years. The hours were calculated by applying the average time recorded against bugs by the development team for each bug priority to the total number of bugs for that priority relevant to scoring.

Figure 4 also highlights the nature of the maintenance effort over time. The number of hours per year for the customizable approach can be seen growing at an increasing rate up until 2006 and then begins to taper off. The shape of the curve for the customized effort can be explained with reference to the number of different scoring implementation done in each year.

2005 saw the introduction of two new scoring models. This is reflected in the tripling of the required maintenance for that year. 2006 had four new scoring models and two and a half times as much maintenance effort required as the previous year. The next two years saw a drop off in maintenance effort. Only one new scoring methodology was implemented during this time.

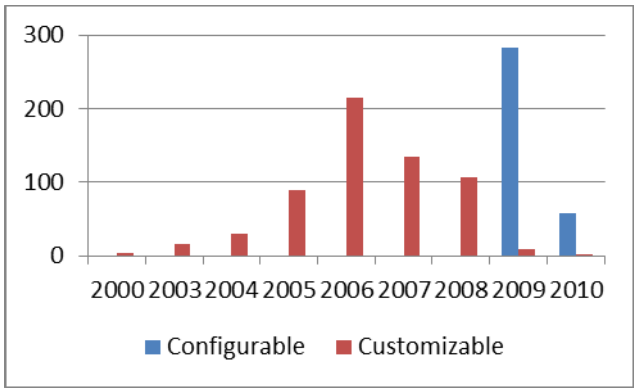

Figure 4. Hours spent on scoring-related bug fixing by year.

The curve for configurable scoring is different. It starts with a huge amount of effort in the first year and then dramatically decreases. The first year maintenance for configurable scoring far exceeded the peak maintenance effort for customizable scoring. As configurable scoring was developed in the first quarter of the year, the next three quarters were spent migrating customers using the old scoring setup. This is where a lot of the maintenance overhead for that year came from. In part it is counted in the initial development cost in Figure 1. Customers also took the opportunity to revise their scoring methodology during the migration and this led to an inflation of the maintenance cost for that year.

2010 saw the implementation of six new scoring methodologies however it did not see a corresponding increase in the number of maintenance hours as would be expected from the curve for customizable scoring in Figure 4. The amount of effort required for the maintenance of fourteen different scoring models is below the 2005 levels of maintenance for only three scoring models. While the figures for the last three months of 2010 are not included in this paper there was no significant change in development effort for that time period.

Overall the required maintenance in 2010 was eight times less than the previous year. This is a remarkable decrease in maintenance effort, even when the inflated nature of the 2009 bug count is considered. The decrease in maintenance hours is also in line with the decrease in implementation time for a new scoring methodology shown in Figure 1. This represents a significant year-on-year reduction in development effort.

C. *Other Effects*

The configurable scoring model was built upon the tribal knowledge developed over the lifetime of Blade. This experience came from implementing scoring models and from talking to prospective customers. It was the bringing together of the different strands embedded in the different customer implementations that allowed the rich tapestry of configurable scoring to exist. The team that designed configurable scoring backs up this point of view in informal discussions. They believe that they could not have designed it without six years of listening to customers and the associated customization of the code to those customers' needs.

Existing customer response to the changes has been positive. They have taken the opportunity to make changes to their risk models when migrating from their existing custom coded implementation to an implementation configured to their needs. Apart from minor look and feel changes, the scoring front end appears to operate the same to end users as it did before. Some customers who use Blade only for the recording of risk events are also considering using risk assessment now that it has the flexibility to match their needs without a large cost attached for customizing the code.

Market response has also been good. Since 2009 the number of customers using Blade has doubled. This increase is not directly contributable to the changes to the scoring model however the ability to configure Blade instead of customize it has helped the company keep up with the pace of sales. During 2010, nine implementations took place concurrently, which is a record for the company. This increase in implementations was not met with a corresponding increase in staffing number or workload due at least in part to the change from a customization approach to a configurable one. Six of these implementations required a new scoring model. One of the new scoring models was very complex and required more than twelve scoring measures. Such a scoring model could not have been implemented without considerable investment if a customization approach had been taken.

V. CONCLUSIONS

ORisk Consulting has made considerable progress in improving the quality of the code in Blade, and in reducing the time it takes to create that code, by adopting core asset

development. The increased quality is confirmed by the reductions in bug numbers across the implementation relating to risk and control scoring. The increase in efficiency is shown by the comparison of initial development times between a customizable approach and a configurable approach that uses core assets. New scoring methodologies are shown to be considerably faster to implement using a configurable approach than customizing the existing code for that methodology. The breakeven point, when the initial cost of building the configurable core asset is taken into account, was achieved in the second year of the evolution after only six implementations.

Using the three tenets described in Section III.A enabled Blade to avoid some of the pitfalls related to SPL development in small development organizations. These tenets -- avoid risk, create developer friendly assets and communicate changes clearly -- encouraged developers to focus on core assets and increase communicate between the different layers of the organization.

Making customers responsible for change within their own implementation lies at the heart of much of the user interface and core asset work that has been done. By giving the customer the power to configure the system to their needs at runtime we remove much of the burden of variability management from development, while delivering added value to the customer. Allowing users within a company to customize their own experience has also had unforeseen benefits. It aids collaboration between users [24], and enables them to use the product as they wish, which is a key advantage in the market.

It is our recommendation that similar sized companies with large variations between customers should develop configurable core assets that can handle existing customer requirements, as well as future requirements. A mature base product can aid the development of core assets as a certain amount of software entropy is required in order to make robust core assets. Mistakes in any industry or practice need to be made before they can be corrected and a better product developed. A good bug recording system is beneficial to all companies in order to identify bug black spots, however with a mature product the trouble areas can be identified by tribal knowledge and the identification of areas of code in which developers dislike to work.

How the approach described in this paper would scale to a larger development organization is unknown. It is unsure how the three tenets that we follow would scale to benefit companies with multiple development teams and projects. There are benefits to companies of any size in avoiding initial risk and encouraging development to build core assets by making development less onerous, however communication between teams and departments becomes more complicated in larger organizations.

One area not handled within this paper is validation of core assets. We have not yet developed a method to reduce the amount of time required for core asset testing. In fact, changes to core assets can lead to the retesting of all implementations of that asset which can be a drain on the resources of the quality assurance team. This appears to be an unsolved problem in the SPL community [26]. Applying the concepts of SPL development to the testing process as we have already applied it to the specification process may result in improvements in this area.

Overall Blade's transition to an SPL has been very beneficial. The increases in productivity and implementation time have been key factors in increasing market share. This confirms that, for Blade, the benefits of SPLs are real. It is hoped that even greater benefits can be gained by continuing the focus on developing core assets and applying our experience of developing an SPL to other areas of our software development cycle.

ACKNOWLEDGMENT

The authors would like to thank Dr. Myra Cohen of the University of Nebraska-Lincoln for reviewing an earlier draft of this paper. Her valuable feedback enabled us to improve it considerably.

This work was partly supported by Science Foundation Ireland grant number 03/CE2/I303_1 to Lero -- the Irish Software Engineering Research Centre.

REFERENCES

[1] Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison Wesley, 2001.

[2] Chu-Caroll, M. Separation of Concerns in Software Configuration Management, In *Proceedings of the ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, 2000.

[3] R.S. Kenett and Y. Raanan, *Operational Risk Management: A Practical Approach to Intelligent Data Analysis*. John Wiley & Sons Ltd. 2011.

[4] Basel Committee on Banking Supervision, *Consultative Document on Operational Risk*, Basel Committee, January 2001.

[5] European Parliament Directive number 2009/138/EC, *On the taking-up and pursuit of the business of Insurance and Reinsurance (Solvency II)*, Official Journal of the European Union, 25 November 2009.

[6] N.N. Taleb, *The Black Swan: The impact of the highly improbable*, Random House, New York. 2007

[7] Krueger, C. Churchett, D. Buhrdorf, R. HomeAway's Transition to Software Product Line Practise: Engineering and Business Results in 60 Days. In *Proceedings of the 12th International Product Line Conference,* pp297-206, 2008.

[8] Staples, M. Hill, D. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, 2004.

[9] Tseng, M.M. and Piller, F.T., *The Customer Centric Enterprise: Advances in Mass Customization and Personalization*. Springer, New York, 2003.

[10] Gang Hong, Deyi Xue, Yiliu Tu Rapid Indentification of the optimal product configuration and its parameters based on customer-centric product modelling for one-of-a-kind production, *Computers in Industry*, Volume 61 Issue 3, April, 2010 pp. 270-279.

[11] Alves, V. Camara, T. and Alves, C., Experiences with Mobile Games Product Line Development at Meantime, In *Proceedings of the 12th International Software Product Line Conference, 2008. SPLC '08*, vol., no., pp.287-296, 8-12 Sept. 2008.

[12] Clements, P. and Krueger, C. W. Being Proactive Pays Off/ Eliminating the Adoption Barrier. Point-Counterpoint article in *IEEE Software*, July/August 2002.

[13] Bosch, J., Software product lines: organizational alternatives, *Proceedings of the 23rd International Conference on Software Engineering*, vol., no., pp. 91- 100, 12-19 May 2001.

[14] Pohl, K. Böckle G, and van der Linden F.J.. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[15] Takebe, Y. Fukaya, N. Chikahisa, M. Hanawa, T. and Shirai, O. 2009. Experiences with software product line engineering in product development oriented organization. In *Proceedings of the 13th International Software Product Line Conference* (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, pp. 275-283.

[16] Tekinerdogan, B. Tüzün, E., Saykol, E. Havelsan A., Exploring the Business Case for Transitioning from a Framework-based approach to a software Product Line Engineering approach. In *Proceedings of the 14th International Software Product Line Conference* (SPLC '10). pp. 251-254, 2010.

[17] Iwasaki, T. Uchiba, M. Ohtsuka, J. Hachiya, K. Nakanishiy, T. Hisazumiy, K. and Fukuda A., An Experience Report of Introducting Product Line Engineering across the board. In *Proceedings of the 14th International Software Product Line Conference* (SPLC '10). pp. 255-258, 2010.

[18] Kruegar, C. Churchett, D., Eliciting Abstractions from a Software Product Line. In *Proceedings of the PLEES International Workshop on Product Line Engineering*, OOPSLA 2002.

[19] Gamma, E. Helm, R. Johnson, R. Vlissides, J., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[20] Beck, K. Crocker, R. Meszaros, G. Vlissides, J. Coplien, J. O. Dominick, T. and Paulisch, F. 1996. Industrial experience with design patterns. In *Proceedings of the 18th international conference on Software engineering* (ICSE '96). IEEE Computer Society, Washington, DC, USA, pp. 103-114.

[21] O'Leary, P. Rabiser, R. Richardson, I. and Thiel, S. 2009. Important issues and key activities in product derivation: experiences from two independent research projects. In *Proceedings of the 13th International Software Product Line Conference* (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, pp. 121-130.

[22] Krueger, C. W. Variation Management for Software Product Lines. In *Proceedings of the Second International Software Product Line Conference* (San Diego, CA, U.S.A., August 19-22 2002). Springer LNCS Vol. 2379, 2002, pp. 37-48.

[23] Pech, D. Knodel, J. Carbon, R. Schitter, C. and Hein, D. 2009. Variability management in small development organizations: experiences and lessons learned from a case study. In *Proceedings of the 13th International Software Product Line Conference* (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, pp. 285-294.

[24] Bentley, R. and Dourish, P., Medium versus mechanism: supporting collaboration through customisation. In *Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work* (ECSCW'95), Hans Marmolin, Yngve Sundblad, and Kjeld Schmidt (Eds.). Kluwer Academic Publishers, Norwell, MA, USA, pp. 133-148, 1995.

[25] Pohl, K. and Metzger, A.. 2006. Software product line testing. *Communications of the ACM* 49, 12 (December 2006), pp. 78-81.

[26] Denger, C. and Kolb, R. Testing and inspecting reusable product line components: First empirical results. In *Proceedings of the Intl. Symposium on Empirical Software Engineering*, pp. 184--193, 2006.

[27] Anastasopoulos, M. Gacek, C. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: putting software reuse in context*. ACM, New York, NY, USA, 109-117.