Software Product Line Migration and Deployment

D. Faust* and C. Verhoef*,**

* *Deutsche Bank AG,*
1251 Avenue of the Americas, 28th Floor, New York, NY 10020, USA

** *Free University of Amsterdam, Department of Mathematics and Computer Science,*
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

`david.faust@db.com, x@cs.vu.nl`

Abstract

We describe a method to migrate multiple instances of a successful single information system to a product line. The deployed product line is able to deal with the over time evolved variants in a cost-effective manner. We proposed and used federated architectures that partition the software into so-called genericity layers. We argue that federated architectures are at the heart of product lines, and we provide compelling arguments as to why federated architectures are a sound weapon in today's corporate strategy: they enable smooth enterprise integration and rapid change. We support our arguments with a real-world case: we successfully migrated a large global transaction and settlement system with many site-specific variations to a product line with a federated architecture. Moreover, we measured the success rate of this architectural modification effort by showing that the annual direct cost savings are in the order of millions of dollars during the deployment of the product line.

keywords proactive/reactive product line, software mitosis, configuration oscillation, grow-and-prune model, federated architecture, genericity layer, relativistic effects of software engineering, release indigestion

Introduction

Traditionally, hardware product lines enable the rapid production of variants of products. This so-called mass-customization can have a strong competitive advantage, for instance, National Bicycle increased its share in the Japanese sports bicycle market from 5 to 29% by moving from mass-produced bikes to mass-customized ones. Instead of a few models, a customer now has a selection of 2 million options for combining size, color, and components. And with an adjustable frame, the ideal measures are taken. In two weeks a fully customized bike is delivered to the customer. Of course, you cannot do this when each bike is seen as a unique product. You need a product line to produce the bikes: in this case computer-controlled welding robots that can handle all the variation points [1].

This can be rather successful, however, in an increasing number of products the information technology (IT) component is becoming the dominant part. Since IT is notoriously late, suffering from cost overruns, and not delivering what was asked for, this also influences the release date and overall quality of the embedded products. For instance, we heard from people working in the automotive industry, that the release date of a new car is sometimes delayed since the software supporting it is not ready. Also cars in operation start to resemble the nature of information technology. A new model car turned out to have a too sensitive airbag system. This is dangerous, so the cars were asked back for a standard repair. In one case we know of, this repair caused another error. To replace the airbag card, the battery was disconnected. For some unknown reason, the motor management processor restarted with the parameters for a smaller engine than actually present. At low speed the car did not expose any problem. But there was: by systematically injecting not enough fuel, the motor will be slowly destroyed. More speed, means more hydraulic power for steering the car. But the power steering routine apparently depended on the wrong cylinder capacity and not on actual speed, so that the necessary hydraulic power was not appropriate for the real velocity—another consequence of the motor management processor that thought the engine was smaller. This resulted in an uncontrollable car at 120 km/hour. We admit that these are the perfect conditions to test in vivo whether the new airbag card would operate correctly. Fortunately, in this case, it did not came that far: the owner, a retired machine designer, inferred in only two weeks from a combination of symptoms and indicators (like a systematic decrease in fuel consumption) that someone or something must have tuned the engine erroneously. The car mechanics told him that they could not even make a change to the entire engine, and they could not find anything. When they connected the car to a diagnosis station, the car owner noticed accidentally that the reported cylinder capacity was 23% too low. When the mechanics reset the motor management processor to the correct cylinder capacity the problems were solved. This feels more like debugging an embedded system, than maintaining a car.

To come to grips with the problems of IT in embedded products like cars, MRI scanners, high volume consumer electronics, etc, the hardware product line idea has been transposed to the software embedded in these systems. Thus the software product line was born (see [2, 3] for general information on software product lines). In some cases a number of unique instances of software are migrated to a software product line. For instance, CelciusTech, a company constructing radar systems on warships, decided to migrate to a software product line approach [4, p. 352] to shorten development schedules, and obtain more uniform systems. There are more examples of migrations like this, but nowadays the majority of the embedded systems, for instance high-volume consumer electronics, start with a software product line right away. We call this approach a *proactive product line*.

In the information systems world we see a different pattern. Information systems start relatively small, and they grow to larger systems when the business can deploy the software successfully to create value. As soon as business starts to use the software, other business prospects emerge leading to updates. When success is spread, so is the software: often such systems are adopted worldwide within globally operating organizations. When the software is in use at many sites, inevitably local modifications are necessary. A local modification can be either a change made by a site, or a site-customized centrally controlled modification. Without proper central governance, only local modifications are made, and the original core system will multiply like a living organism, and after a while a significant number of variants of the system will emerge. We call this phenomenon *software mitosis*.

For large information system owners it is not uncommon to fall prey to software mitosis. There are a few common causes:

- increasing customization;
- local control and modification;
- mergers and resulting consolidation;
- geographical distribution of the organization.

For instance, the US department of defense possesses myriads of similar payroll systems and many variations of personnel training systems, and there is more redundancy [5]. In these situations, the crucial question is: How to bring all this variation under control? For all of the above cases, the ideal solution is the same: turn the existing variants into one overall system where the points of variation are migrated into a product line. We call this a *reactive product line*. Both proactive and reactive product line approaches are extremes and reality is usually somewhere in between. Often, even when starting with the green field to develop a proactive product line, this evolves over time to a structure that is quite different from the original. The forces of increasing customization for a growing user base, of local control and modification, and of corporate mergers and resulting consolidation are very likely to result in significant architectural alteration. So whether you are to deal with much historical evolution and accident (i.e., reactive case), or started with an explicit and clear architecture before encountering the aforementioned forces (i.e., proactive case), or you are in a hybrid situation somewhere in between, the methods we describe in this paper are very likely to be useful.

Configuration oscillation

Software mitosis should not be viewed as a bad phenomenon, because apparently the business recognizes the value and usefulness of the software and the business and development team work together to make it all the more flexible and useful. What we must do is to architecturally manage its negative consequences. In the short term, there are huge demands to deliver software for specific clients, because the clients want to capture new business and the developers want to support them as best they can. Overall, these demands are met quickly with specific implementations, often including code cloning. Over time, say 2–3 years henceforth, the developers find themselves in an untenable situation, because the growth and compounding of specific changes make future changes even more difficult and costly. Sooner or later costs and efforts will inflate proportionally to the rate of growth of the code variants. Development then responds by stating they cannot support this situation and that a generic system is needed. It may even be that, after development presents the problem, the business users agree to the effort that there are now funds to consolidate and thus conduct the major project to move to entirely generic mode. So the developers then feverishly work (perhaps for many months) to create the generic system by consolidating changes into core code units. Now, after this supreme effort, the situation is under control for the developers but their user base is still growing, and the growing demands still must be met. What they now find is that the fully generic code makes rapid-fire changes difficult to coordinate, that it increases time-to-market, and that it causes unwanted functional rippling and other side-effects for their demanding clients. Then, a cycle begins again, because the site-specific forces are once again too great, and the system over time becomes specific in response to these forces. But they later incur unacceptable operational costs, and so need to consolidate, and so on. We call this phenomenon *configuration oscillation*. Thus, the software oscillates between being largely generic and being largely specific—the development team is doing their best to optimize against their changing user base and demand level. It often follows that the core configuration becomes infeasible once the usefulness and global demand for the software substantially increases.

If your software system has moved well up the maturity curve, you will need something different. You need to find a way to make costs and efforts reasonable, yet preserve client satisfaction—a new and selectively hybrid solution.

The grow-and-prune model

How shall we then deal with the mitosis, the configuration oscillations and their consequences? An everyday model represents a key concept in our strategy. We like to compare constructing software with growing a tree in a special shape—be it a bonsai tree, or a tree in the Swiss tone wood destined to become a fine sounding board for an expensive piano. Without both allowing growth and doing the proper pruning, the goal will not be reached. We believe the tree should always have several layers: the main trunk as the global layer, the side-branches can form the regional layer, the twigs form an even more specific layer, and the leaves form its most specific layer. Software mitosis is like letting the tree grow freely without pruning, i.e., without architectural governance. Quite naturally, branches, twigs and leaves may spring and grow, but the key here is to maintain the right form and balance and thus optimize robustness, process efficiency and growth in the desired fashion. Capping mitosis entirely will prevent growth, like constant or excessive pruning will kill any living organism. So we must allow some mitosis. Likewise, configuration oscillations will ultimately kill, or at least stunt the growth of, your software system. The key is to find the right methods and degree of governance for these phenomena. This governance is maintained best with our grow-and-prune approach. Extending the grow-and-prune metaphor, we should have a growth season and a pruning season. We enable the business to fertilize new source trees to grow new business opportunities, and with part of the returns we justify the sometimes significant costs of pruning the system into the right generic-specific blend. We thus reduce complexity in maintenance, simplify release management, and enable faster and more directed growth.

Technically speaking, this is not so easy. From the configuration point of view, there is initially no source tree, but instead a *source forest* containing the actual variants in production, perhaps several enhancement projects on various genericity layers, or several so-called *playpen branches* to deal with special requests and to try out new business opportunities. From this we suggest on a semi-annual basis the consolidation of the concurrent development across the source forest into a single tree, from which only valid production variants are inferred. If the combination of development ordering, business need, or enhancement projects makes it necessary, you can create additional branches with known limited life spans with the intent to consolidate them on at least a semi-annual basis. If possible we order the projects, to limit forest growth and to minimize the inevitable rework. And, as we will soon see, there are unexpected effects of our software engineering methods, for example, that some degree of code redundancy leads to lower net impact on the entire development, deployment and acceptance process (that redundancy is documented, of course, to limit rework).

Federated architectures

A characteristic of software product lines is that their design is driven by so-called variation points: the parts of software that are to be flexible, so that variations of the system are supported by a single source with multiple instances. What happens with product line development is that various *genericity layers* are being identified, and that the software architecture reflects these layers. We call an architecture that is designed in this manner a *federated architecture*. Note that our use of the term federated architecture is different from the one used in distributed, heterogeneous databases where it is used to characterize techniques for integrated access of a number of such databases [6]. We use it to indicate the degree of business impact to parts of a system. If the business impact of some functionality is bound to one use-site, then also the technical and managerial impact should be. If the business impact is larger, then also the technical and managerial impact should be larger. Only global business impact should have global technical and managerial impact. A federated architecture aligns this by keeping genericity with respect to business impact aligned with technical and managerial impact. In a proactive product line you identify the genericity layers up-front, and in the reactive situation, you have to reconstruct the genericity layers from multiple instances of one system or from similar functionality in different systems (e.g., after a merger).

At least three genericity layers should be present in product lines: the *global layer*, the *regional layer* and the *local layer*—whence the name federated architecture. In fact, layers need not be just three: we have seen more layers in systems supporting PBX switches. Our running example for this paper stems from the information systems industry. It is a successful global trading and settlement system (GTSS) that is used worldwide at Deutsche Bank. The global layer of a GTSS contains the functionality that should be available at all use-sites; think of SWIFT messaging, common date, currency, and calculation routines, menu functionality, and system installation parameters (SWIFT is short for Society for Worldwide Interbank Financial Telecommunication). The regional layers contain variations of corporate action functionality, currency cash account handling, or repurchasing agreement functionality, as required for only certain lines of business: custody banking, private banking, or investment banking respectively. For private banking, think of interest payments or fees on credit and debit positions, and for investment banking a typical example is the short-term repo trade processing (repo stands for repurchase agreement). Local layers contain further variations on these functions and perhaps specific regulatory reporting or specialized interfaces or data feeds required at a specific site. The system measures about 60000 function points, contains thousands of code units, and has many instances. There are variations for different banking services in many countries, and in some cases multiple varieties even within a country. In our case, this amounts to more than 15 large to major installations of the system and many more minor ones, and therefore deployment costs of this entire living organism easily inflates to many times the cost of deploying the original system. This pattern is similar to embedded systems like MRI-scanners or PBXes: each site is unique, and also for these systems software mitosis needs to be brought under control [7].

In the software mitosis phase, often regional (intermediate) layers are not present, or if they were once envisioned by the original software architect, they are structurally underutilized. The reason for this is managerial: who is going to pay for such a project? Obviously in an existing situation where software mitosis is not recognized, there is no sponsor for projects that belong to the regional layer. But even if there is, it is often difficult to coordinate regional layer changes, due to lack of interest from the sites, for instance. Just like leaves want to grow, without being bothered by long-term consequences. This leads to bloated site "specific" software, and for those changes that were not done locally, global development will add functionality of unknown genericity to the core system. We list some of the long-term consequences that we experienced in this kind of situation.

- No one knows anymore what *the system* comprises. It consists of an uncontrolled set of variations of components and parts that run at each use-site.
- Ownership of certain parts of this off-spring becomes unclear, up to the point where global development has no longer access to part of the sources anymore. This can be due to local ownership wars, or a consequence of outsourcing "local" maintenance to a service provider. Note that when parts of such multi-version systems are outsourced, it becomes much more difficult to cap mitosis, let alone to migrate the off-spring to a product line.
- Release management of such systems becomes error-prone and prohibitively expensive.
- Configuration management of such systems gets out of control.
- Deployment costs of the sum of the variants inflates: instead of one system a significant number of similar but unrelated variants needs to be continuously updated leading to unacceptable corporate costs.

Enterprise integration

In addition to the long-term effect of getting and keeping the software under control, with its substantial benefits in terms of increased efficiency, cost control and reliability, there is also a very compelling strategic reason why this is a highly undesirable situation: it hinders enterprise integration. When you decide to merge with an enterprise or acquire a company to complement your services or products, you have a business advantage if your crucial systems have a federated architecture. Why is this the case? Since virtually all business is heavily supported by IT, IT has grown from an internal service cost issue to a strategic company asset—but not many leading companies recognize this [8]. Managing those assets is therefore of crucial interest. If you have a federated architecture, you *actively* know what your generic business processes are, what your local processes comprise, and all the genericities in between.

Before the ink of merger contracts is dry you will face an enormous problem: how to integrate all these variants into a flexible configuration? A company that has federated architectures in place for their most critical systems will have a strong competitive advantage since integration of another business critical system to an already federated system is consciously enabled. Moreover, once the merger is in progress, active strong knowledge will likely make this system the dominant one, perhaps even tantamount to functional scope and technical platform considerations. When Deutsche Bank merged with another enterprise, there were several systems evaluated against one another with the aim of carrying a single system forward as the trading and settlement system for the merged company. In today's world where software takes complete precedence over hardware in the cost and effort contest, and systems are being compared for scope of functionality, technical platform and robustness, the system being actively managed with a federated architecture has a marked advantage. For instance, the GTSS system that we equipped with a federated architecture won the contest in a merger. Despite superior functionality at some points, and better underlying technical infrastructure, the other candidate systems were just not fit for rapid-yet-controlled global growth. So having a federated architecture with the necessary layers is of course not the total enterprise architecture solution, but clearly positions a development group and its software for enterprise-level activities and is thus a cornerstone of the enterprise integration.

Summarizing, federated architectures enable enterprises to integrate and compete more effectively and efficiently. But how to get towards such an ideally-sounding future of thoroughbred systems from the current situation where some of your crucial systems grew unfettered and were accidentally crossbred throughout the company?

Layer distributions

Obviously, no one has federated architectures in mind when a single-instance system is envisioned. Also the idea that such a system would become a *mother system*—breeding variants as success spreads—does not come to mind. The dangers of software mitosis are rarely recognized in this stage, and are what you could call *relativistic effects of software engineering*. We use the latter term when the size of software becomes so large that classic thinking about software systems breaks down, just like Newton's laws in physics break down when velocity becomes very large.

We have developed a method to detect genericity layers in such off-spring, based on six metrics. Using our method, the layers become apparent. For each descendant code unit we can decide in which layer it should reside, and then can carry out a systematic redistribution of the code units. This effort migrated a multi-version system into a product line where variation is controlled by a federated architecture with code units allocated to the appropriate genericity layers. We applied this approach to the aforementioned global transaction and settlement system (GTSS) that had fallen prey to software mitosis—due to its success for the company. In the case study we consider, the direct benefits were in the order of millions of dollars in cost savings, and the indirect and inferred benefits were substantially higher (as we will show later on).

Phase	Global	Regional	Local
I	61%	25%	14%
II	55%	35%	10%
III	29%	58%	13%
Total net impact	-32%	+33%	-1%

Table 1: Genericity layer realignment.

Our effort consisted of several phases (see Table 1). Phase I included a system cleanup, which is a first step into a standardized federated architecture. This amounts to elimination of obviously redundant library entries, deletion of other redundant parts and the creation of a new release exempt from these irregularities. Phase II consisted of the analysis of the cleansed system using the six metrics that we will discuss in detail shortly. In phase III we interpret the accumulated metric data and decision rules we developed, and migrate the majority of the code units accordingly. The result of phase III is a release of a product line comprising all the existing instances based on a 3-layered federated architecture. We note that for systems of lesser consistency and quality than this system, more prerequisite work is necessary. For instance, when the systems are from different companies due to a merger, they are likely to be disparate, and then you may need gap-analysis, rearchitecting [9, 7, 10, 11, 12, 13], dialect and language conversions [14], platform migrations, code unit cleanups—think of dead code removals and the extinction of unstructured coding [15, 16]. After these efforts, a higher homogeneity is achieved and only then you can start to apply our approach.

In order to give an idea how the various layers are instantiated in a large software system, we present the original layer distribution for two of the libraries used in our running example (see Table 2). These libraries together consist of 3500+ code units.

Genericity	% of units library 1	% of units library 2	Total % of units
Generic core	62%	56.9%	61.1%
LOB1	18.6%	16.9%	18.2%
LOB2	5.8%	10.1%	6.9%
LOB3	none	none	none
site-1	1.0%	1.7%	1.2%
site-2	1.0%	0.2%	0.8%
site-3	0.1%	0.4%	0.2%
site-4	0.7%	none	0.6%
site-5	1.1%	2.1%	1.4%
site-6	1.5%	1.6%	1.6%
site-7	7.4%	9.8%	8.0%
Total	99.2%	99.7%	99.9%

Table 2: Original genericity layer distribution on a 3500+ code unit sample.

Table 1 outlines the net-impact of the reallocation of code units from the original layering as depicted in Table 2. The total net impact is a 32% decrease in the global layer, a 33% increase in the regional layer and a 1% decrease in the local layer. As can already be seen from the accumulated data, indeed the lack of an organizational layer in a company induces a problem with the sponsoring of projects with a regional impact. This situation leads to either globally funded or locally funded projects. But even if the organizational framework is in place, it is hard to explain genericity realignment projects, yet they are architecturally crucial.

Based upon the accumulated data it seems that almost nothing changed to the local layer: only 1%. Therefore it is illustrative to realize that in phase II the local layer first decreased with 4% and later on increased with 3% due to considerable movement from global to regional, regional to local layers and vice versa. So, in general there were more movements from layer to layer, because we ensured each unit ended up in its proper layer, and thus the net total movements by layer actually underrepresent total movements. But each of these movements is in fact crucial. The code meeting the evaluation criteria has replaced the code not meeting the criteria for that given layer—with key resulting benefits.

As you can see in Table 2 the libraries show a disproportionate tilt in the direction of the generic core residing in the global layer. The historical reasons for this tilt are that the configuration had become much too specific, and that there had been a major oscillation response bringing it to great genericity. But there was too much growth in the user base, and too much overall demand to carry on with the heavily generic configuration. And, although a middle layer had been created in recognition of the oscillation problems, this layer was severely underutilized—one line of business (LOB3) did not have any regional code units. This was revealed after we applied a set of metrics which clearly suggested a better use of the layers and a better balance of those layers.

Metrics making our life easier

In the context of this large system, we have to deal with code duplication, impact to business users, control, efficiency for the software developers, and the complexity of maintenance. We wanted to find the most pragmatic measures for this goal, and then take a holistic view of them. We first summarize the metrics and then treat the less obvious ones in greater detail.

- Utilization (U): measures the actual use of the functionality residing in a physical code unit.
- Volatility (V): measures the frequency of modification of a code unit.
- Procedural Code Size (PCS): the well-known metric returning the physical number of lines of procedural code for a given code unit. If you are using an OO language, then PCS measures the lines of code for the nonabstract methods.
- Specificity metric (S): the amount of branching over and percentage of conditional logic in procedural code that meets the business requirements for a particular line of business. We measure the functional variation by parameterization. In case you are using an OO language, the specificity metric measures the degree of potential polymorphism.
- Cyclomatic Complexity (C): the well-known complexity metric developed by Tom McCabe [17].
- Mitosis delta (Mδ): this measures the code difference (or delta) between multiple occurrences of the same code unit.

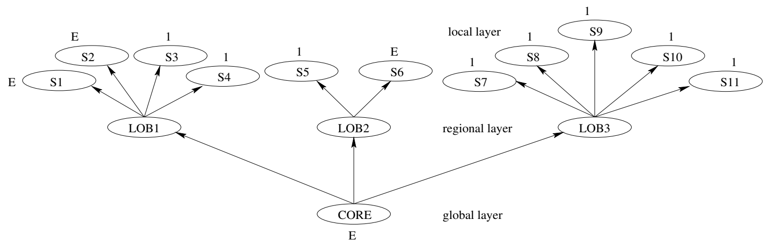

Figure 1: Calculating utilization for code unit `PROG`.

Utilization This is a metric that describes the extent to which a code unit is actually used across the existing genericity layers, as reflected in the build structure. The intended interpretation of the utilization metric is that low utilization should be found in local layers, medium utilization in regional layers and high utilization values in global layers. For each code unit we can calculate the utilization metric by using the build structure. We see the build structure as a graph (see Figure 1) in which we mark all the nodes in the build graph with an **E** if the code unit is present in that location of the build graph. We also mark all the leaves of the graph (which correspond to the local layers) with a **1** if this leaf uses the code unit but does not contain the unit itself, i.e., there is no **E** at that node. Existence of another code unit variant for that site nullifies any utilization for the code unit being measured. Then for the layer under the leaves, which corresponds to the regional layer, we add the **1**s and divide this by the total number of leaves of that node that do not contain the code unit (so without an **E**), and we assign that number to the node. This is reiterated for intermediate genericity layers if they are present. Finally, for the root of the graph, corresponding to the global layer, we calculate the utilization metric by summation of the nodes one level higher. Figure 1 shows how we determined the utilization value for code unit `PROG`. In the build graph we marked its actual existence with an **E**. We also marked the sites where this code unit is used, but not present in the local layer of the system (it is called from the regional, or global layer). At the line of business node (LOB1) we then accumulate $2/4$, at LOB2 we calculate $1/2$, and for business line LOB3 we find $5/5$. Overall, for `PROG` we find $U = 2/4+1/2+5/5 = 2.5$. We first calculated this metric semi-automatically but in phase III an automated logging and calculation mechanism was incorporated to generate the utilization value for each code unit of the system. We also used automated periodic tracking of the invoked programs at production sites to to check their actual utilization.

Volatility This is a measure for the volatility of a code unit. It measures the historical and expected changes to a code unit over a two year period. The historical volatility (V_h) of a code unit was obtained by simply reviewing the change log within the code unit over a period of 12 months. This number is automatically retrieved by the source configuration and versioning system that is in place for this GTSS. The expected volatility (V_e) is estimated with respect to open issues and planned projects that will clearly impact the code unit over a period of 12 months. Since V_e is less accurate, the weightings of historical volatility versus the expected volatility were chosen in order that empirical data took precedence over planning judgement. We calculate the volatility of a code unit as follows: $V = 0.7V_h + 0.3V_e$.

Specificity metric This is an application specific metric expressing functional variation by parameterization. That is, depending on the business goals, some issues are more relevant than others, and based on that we can define the application specific metric (you could use the GQM approach [18] for that). Let us give a few examples that illustrate how to develop a sound specificity metric. The GTSS supports transactions and settlements of financial products. To decide on specificity, we used a combination of three application specific aspects: we looked at multi-currency code, multi-system installation code, and feature table code. A script was created to target for this type of branching code, by identifying the conditionals and counting the lines of associated branched or featured code. The script was supplemented by manual inspection to prevent false positives. We calculated the specificity metric as a percentage of branched and featured code of the total number of lines of procedural code.

For other application types, other specificity metrics can be developed. For PBXes this can be branching on system installation, billing type, and country. For commercial software this can be branching on versions, language, country, etc. For integrated radar systems on war ships this can be branching on ship class, defense or attack type, language region, country, and so on [4].

Mitosis delta Software mitosis leads to duplicates and near duplicates. For each code unit we identified their number of occurrences, and using the source code configuration management tool for this GTSS, we calculated the source differences percentages for code units with near clones.

Why these metrics?

The *utilization metric* indicates the actual usage across the current genericity layers. Of course these layers are not designed as such from the beginning, so code units can have a utilization that is not in accord with their presence in a certain layer. The lower the utilization value, the greater the candidacy for a code unit to reallocate it into the regional and local layers. The aim here is to deliver to the regional and local layer only those code units that are actually used by those layers. Only code units with a very high utilization should be present in the global layer. In our case $U = 3$ is the maximal utilization (since in the GTSS three genericity layers was appropriate). Overall, code unit `PROG` has a high utilization (see Figure 1). Sites S1, S2, and S6 depicted in the source tree of Figure 1 are all in one country. And all these sites use local-layer versions, the other countries all use the global version. So, there is code duplication in this country, to prevent other sites being bothered by their country-specific updates. But in fact we can see that `PROG` needs refactoring [19]: the global-layer version should get an extra call to a local-layer routine that is specific for this country. Then the duplicates of `PROG` can be removed from the production source tree, and instead truly country-specific code is put in the local layer. You can see that we could have opted for an extra layer: a country-layer. Given the number of total sites, this was not necessary. But in the PBX switch industry, where the number of installation per country can be substantial, a country layer is justified.

The higher the *volatility* of a code unit, the greater the candidacy for code unit allocation to regional or local layers. The aim of this metric is to reduce the frequency of impact to business lines or use-sites of the system *not* requesting a particular change. Apart from the genericity layer (re)allocation purpose, this metric is also beneficial in identifying code units that need special attention. Volatile code units should be properly encapsulated and carefully interfaced. If the local version of `PROG` would never change, there would be no reason for local copies. But this local version is volatile. That is why there are local copies, not to bother another site in another country with an update only relevant for this country. And this metric gives another hint that factoring out the country specific code could be necessary.

The greater the *specificity* of a code unit, the greater the candidacy for code unit allocation to regional and local layers. We look at branching structures that distinguish cases based on business-specific functional variation by parameterization. In our case this is branching over the values of system installation on geographical location or region, branching over the base currency defined for the location or region, or branching over the feature table comprising a site profile listing various functionality, driven by parameter values. Then we look at the code volume to which the branches point. If that portion is very large, it is an indication that the code unit must be specific for a certain application-specific aspect of the system. If there is a lot of branching, but the code volume of the specific code is not too large, we are probably dealing with a code unit that belongs to a regional layer. For instance, it can be a unit that points to site-specific code units, and serves as an abstraction layer for a line of business (viz. an abstract class in object-oriented programs [20, 21, 22, 23]). If the branching is low, or somewhat higher and the code volume of the specific code is very low, the entire code unit consists probably of generic code, and it should perhaps belong to the global layer. Yet, this decision needs also to reflect on outcomes of the other metrics.

We are interested in McCabe's *complexity* as well, since we want to have a stable, simple and maintainable global layer. In other words a thin kernel. But we also do not want to have too much complexity in site-specific code units, because then multiple occurrences of complex code will end up in the system. The replicated complex code units will inflate maintenance costs. So, complexity is too risky in the global layer, and too expensive in the local layers, whereby the higher the value for complexity, the greater the candidacy for code unit allocation to the regional layers. The local versions of `PROG` remained fairly complex after refactoring, so according to this metric it is better to move the local code unit one layer down.

The *mitosis delta* informs us on the degree of uncontrolled growth in the system. If versions of the same code unit exist in different genericity layers, and the mitosis delta is near 0% we are dealing with pure clones. Depending on other factors we can remove them or leave them in place. For instance, when identical code units with multiple occurrences in the sources reside, superfluous units might need deletion, and if so, calls to the removed code units should be replaced by updated singular calls. For the near clones, with a mitosis delta less than 10% we may consolidate the more specific-layer version with the more generic-layer version. We could refactor the differences into subroutines and delete the more specific layer occurrence. When we do this, all the references to the old code units should be updated in the make facility of the system. But again, this depends on the values of the other metrics.

Genericity re-layering

Some of the above metrics give ambiguous indications on what to do, or come up with conflicting data. Just think of `PROG`: according to two metrics it needs refactoring into a global and regional part, but in keeping with the advice of McCabe's metric, the local part should reside in the regional layer. A list of metrics alone will not give you the decision tool to allocate code units so that a federated architecture with true genericity layers emerges. To shed light on these aspects, we examined the gathered data in great detail, while keeping in mind the current distribution of the code units. We looked for patterns for each metric across the current genericity layers.

Relations and conflicts

We considered the relations between the chosen metrics as we reviewed the code units. As a result, we came up with Figure 2 that indicates general directions for moving code units. But this figure is not the complete picture, as `PROG` indicated. Moreover, we have to deal with the relativistic effects of software engineering. In small systems you can use simple rules like: there shall be no code duplication. Then a metric indicating code

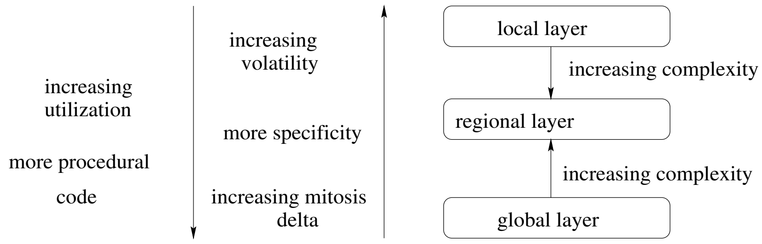

Figure 2: Summary of suggested layer movements directions

duplication leads to only one decision: remove the duplicates. When dealing with a multi-version 60000 function point GTSS, not minimizing on code duplications indeed inflates complexity of maintenance, but forbidding code duplication leads to even more cost inflating, and unacceptable business impact since then more releases are necessary than with some code duplication. There will need to be trade-offs.

How then can we better understand or interpret the not so clear or even conflicting combinations? One option is to combine the metrics in one overall metric by a simple linear combination with statistically-inferred weights. Such a formula would take the form:

$$T = aU + bV + c\text{PCS} + dS + eC + fM\delta$$

The problem with this approach is that with respect to the often conflicting correlations, two antagonistic metrics will easily cancel each other out; obviously not leading to concrete guidelines for allocating code units to the correct genericity layer. Because of these negative correlations, we cannot consider the evaluation criteria in a simplistic, say classical way, but must do so in a relativistic fashion. We need to consider multiple dimensions simultaneously. The metrics serve as aids for making difficult holistic decisions to establish the optimal configuration balance. Therefore, we need a different approach than taking a weighted average.

Layer	*The six metrics*					
	U	V	PCS	S	C	Mδ
Local	L	H	M,L	H	M,L	H,M
Regional	M	M	H,M	M	H,M	M,L
Global	H	L	L	L	M,L	L

Table 3: Evaluation table.

First we constructed an evaluation table (see Table 3). In this table, **L** is Low, **M** stands for Medium, and **H** means High. We give an advice for each metric, and each genericity layer, on what to do as if there were no conflicts. We assign thresholds to each metric. For instance, a utilization ranging from 0.01 to 0.59 is low, and therefore the advice is to allocate code units within that range to the local layer. Code units that fall within the 0.60 – 2.69 interval are to be allocated to the regional layer, and from 2.70 and beyond the code unit should be in the global layer. However, code units with a utilization of zero, should be deleted completely, and calls to such units should be removed.

Layer	*The six metrics*					
	U	V	PCS	S (%)	C	Mδ (%)
Local	$0.01 - 0.59$	≥ 16	≤ 3500	≥ 10.0	≤ 400	> 8
Regional	$0.60 - 2.69$	$15 - 6$	≥ 1000	$9.9 - 5.0$	> 400	$3 - 8$
Global	≥ 2.70	< 6	≥ 2500	< 5.0	≤ 400	≤ 3

Table 4: Thresholds for the six metrics.

We summarized the various thresholds in Table 4. The thresholds are pragmatic in the context of the particular GTSS. We expect some variance from application to application depending on the particular implementation of the system. For instance, the thresholds of the utilization metric are dependent on the actual build structure. Other systems presumably have other build structures, so we expect the architects of another system to reinstantiate this metric. We explain a few actual values that are specific for our example system (the GTSS), to give you an idea of what these values mean.

Suppose that a code unit has a utilization of 1.20. This means that the code unit is used by all sites of a line of business and at least at one other site of another line of business. Or it is used by half of the sites in two regions and at least one other site for another region. Note in these examples, no other code unit variants existed. In our situation this is a medium utilization of the code unit, which we expressed by the M in Table 3.

A specificity of 7% for a code unit means that, based on the review of the conditionals and parameterization in the code unit, we have identified that 7% of the total lines of code may be executed based on the values of system installation on actual location, the base currency defined for that location, or the feature table comprising a site profile listing various functionality, driven by parameter values. This is essentially the part of the code unit that executes differently per site. If the specificity grows above 10%, the code unit should be pushed to the local layer: too much of the code is now locally-aligned, and this makes for increasing difficulty of modification and increased risk if used in the regional or global layers.

For each code unit we accumulated the six outcomes of the thresholds in a six character string like GLLRGR, where the **G** stands for Global, the **R** for Regional, and the **L** for local. For these strings we constructed a permutation table with an advice on what to do with a certain six character string, and thus where to place the unit even when individual metrics may indicate differently. We give an impression of this in Table 5. Note that this table contains potentially $3^6 = 729$ entries, since the order of the strings matters.

There is not necessarily a strong correlation between any of the metrics—and no common root driver. Any reduction exercise therefore leads to potential erroneous decisions. With an eye to all 6 metrics we developed simple decision guidelines. They are listed in descending priority.

- Overrides for standardized functional groups. By this we mean that code units that implement certain standards, such as SWIFT messaging, date calculations and formats, units of measure, etc. will always be considered for maximum genericity even if their metric valuations indicate otherwise.

- We adopt the rule of the majority, e.g., 4 Gs override 2Rs.

- Utilization and volatility are dominant metrics, and they can serve as tie breakers. For instance, you cannot afford high volatility or low utilization in the global layer.
- In case of huge variance, like 2 layers between metrics levels, these should go to an immediate action list for further inspection. For instance, one metric indicates reallocation to the global layer, and another to a local layer. Such a code unit could be a composite of local and global code (e.g., `PROG`) and may need refactoring.

GGGGGG	$\rightarrow$	G
GGGGRG	$\rightarrow$	G
GRRRRR	$\rightarrow$	R
GLLRGR	$\rightarrow$	L
...	$\rightarrow$	...
LLLLLL	$\rightarrow$	L

Table 5: Permutation table.

Architectural gardening

While the word architect stems from the ancient Greek words $\alpha\rho\chi\eta$ and $\tau\epsilon\kappa\tau\omega\nu$ literally meaning the *main carpenter*, we like to think of an architect as the *main gardener*. Now that the metrics are defined, and their relations and the nonintuitive, say, relativistic aims are considered, there is the final responsibility of the architect: letting some code units grow, while pruning others. This gardening is a true architectural modification effort: the code view of the system is sometimes drastically modified [11]. You must carry out such an effort with the support of the best domain experts of the system. Domain experts are required because only they understand the subtle differences between code unit versions, and the needs of the users as they evolved historically—so are best equipped to prune and consolidate. A source tree will become increasingly stable as it moves along the maturity curve, and its trunk and major branches will vary less and less over time. However, we do not want to inhibit new growth, and instead want to ensure that new growth areas can emerge supported by fertile business propositions. So, with the first genericity realignment you need to stringently define the trunk and major branches, yet possibilities will remain open to re-layering over time. Regional branches can later move to the global layer, or local branches to the regional layer. This can also mean removal of code units if the business proposition did not create enough value. The key is that we want to give each code unit the chance to grow with higher volatility, until it becomes eligible for proper allocation with time. The table thresholds and evaluation process need to cater for this.

In this way we could successfully assign a genericity layer to the majority of the code units. Of course when you do this for thousands and thousands of code units, there will always be exceptions. The interesting cases are those that are in the twilight zone: the regional modifications for which it is hard to find sponsors due to lack of managerial structure or agreement (caused by lack of interest, or too many stakeholders). To track such cases we analyzed the current and proposed code unit distributions. We inspected the actual values of the metrics and the code units, and designed refined rules that take care of such special cases. We give a few of these and other rules that further help reduce the potential 729 different outcomes of Table 5. Note that we use actual values that are context-specific for our running example. For other systems, other values are necessary.

- Global code units with a high specificity value that are reallocated to a local layer that have a utilization of more than 2 and a complexity of more than 275 should be allocated to a regional layer.
- Global code units with a high volatility value that are reallocated to a local layer that also have a utilization of more than 2 and a complexity of more than 925 should be allocated to a regional layer.
- Global code units with low utilization and medium volatility that are reallocated to a local layer that additionally have a complexity larger than 2300 should be allocated to a regional layer.
- Global code units with specificity larger than 25% that are allocated to a local layer but that also have a utilization higher than 2.7 should stay in the global layer.
- Global code units with volatility between 8 and 10 that are allocated to a regional layer and that have low complexity and high utilization should stay in the global layer.
- Regional code units with medium volatility and medium specificity that are allocated to a local layer and additionally have a complexity higher than 1600 should stay in the regional layer.

Using these refinements to the evaluation table, we could assign nearly all the code units to the appropriate layer. Of course, then still there are a few exceptions. Those issues cannot be simply solved with a metric, a table, or some specific rules. To get those code units resolved you need actual knowledge of the system. There were about 0.5% of the code units that needed this type of treatment. We give one example. When the metrics are giving really mixed messages, it is most probable that the code unit is not appropriate. A code unit with a utilization of 0.8, a volatility of 18, procedural code size of 2000, specificity of 3%, complexity of 1000, and a mitosis delta of 2%, cannot be easily placed. This code unit consisted in part of regional code and for the rest of site-specific code. It was first refactored, and as a consequence cut into its regional and site-specific parts. After this effort it became easy to re-evaluate and then allocate each part to its most appropriate layer. Remaining borderline cases should be erred to a more specific layer than to which it was allocated, letting the code unit live up there until the next pruning season.

This genericity realignment process is conducted every six months, incorporated in the bi-annual major release of the system to all sites, so that the undesirable growth is pruned while growth in the right direction is consolidated. We believe this evaluation, pruning, and migration process leads to the optimal balance between code duplication, impact to the users, control of change, and efficiency of the developers with respect to complexity and maintainability for successful deployment of the product line.

Cost and Benefits

No development group will directly score points with its users by performing a genericity realignment. Users in general view a system as their own, or at least want to, and so this work is purely architectural and will be difficult for your user community to understand. It is nonetheless crucial, and we will illustrate what benefits both the development and user communities will miss by choosing not to employ our methods. The metrics used so far reveal nothing to corporate management, since at the executive level there is only one metric: dollars. Therefore, it is crucial to be able to estimate and project benefits, and ultimately to measure the actuals.

	C/B 2000 ($)	C/B 2001 ($)	totals ($)
Cost of realignment	-117600	-117600	-235200
Reduction of code unit redundancy	885200	1097300	1982500
Reduction of average build costs due to smaller build sizes	249000	381900	630900
Fewer build fails	122400	212000	334400
Reduction in development test costs	391500	516600	908100
Reduction of release digestion costs	2140000	2140000	4280000
Totals for 2000 and 2001	3670500	4230200	7900700

Table 6: Cost/benefit summary for 2000 and 2001.

In Table 6 we summarized the cost and benefits of deployment costs and benefits over 2000 and 2001. On average, this amounts to about 4 million dollar per year in cost savings. The annual budget for a 60000 function point system is such that these savings are in the order of 3–7% of the total annual budget. This may seem low to the uninitiated, but it is not realistic to assume that a single technology can reduce costs with an order of magnitude, as found by Brooks and eloquently summarized: there is no silver bullet [24].

Cost of realignment

Before we were able to deploy the product line, we had to set up genericity realignment to migrate to a product line approach. This incurred the following costs. We started with a 4 person-month compilation of data into spread sheets. Then, we needed 2 person-months for defining the metrics and methods. An additional 4 person-months (2 months, 2 persons) for support were necessary. Furthermore we used, 3 person months (1 month, 3 persons) for inputs on utilization, specificity, and manual interpretation (e.g., help from business line developer teams). Then 1 person-month for McCabe's complexity, code instrumentation, and compilations was necessary. The make facility was already in place, but we needed to alter the build structure slightly, which took an additional 2 person-weeks (if you are lacking a make facility, it might take easily 2 months to create it and verify it). The overall cost for this phase was: 14.5 person-months, so with 200 working days per year, this amounts to 240 person-days. To assign a dollar amount to this, we will use an average (but ficticious) fully loaded developer daily rate of $1400, which is the average rate being paid in North America (see for instance [25] or [26]). So this leads to an initial investment of approximately

$336000. An investment of this magnitude was done before 2000 which was paid back in a few months on cost savings for testing alone. We refrain from details over 1999, but focus on the next two years, where the operational costs of genericity realignments and the deployment benefits were becoming more and more apparent.

Actual realignments take one person-month; the review and re-testing of the full product takes another 1.5 person-months. Overall, these costs are 2.5 person-months, or 42 person-days, costing $58800, which is a marginal extra cost when a full release of the product is due anyway.

Code unit redundancy

In phase I we identified code units that could be deleted as a result of our cleanup operation. We gathered historical data regarding the average rate of change of code units in general and established a running rate of change against the set of deleted code units for *after* deletion. Then we established a conservative average for the development time spent per code unit changed (using a linear rate of change), and we projected the cost savings as soon as we deleted the code units. Note that you can never come up with *actual* savings after deletion since deleted code units are by definition no longer modified. Then after the cleanup phase (phase I), we accumulated metric data in phase II that we used in phase III to establish a federated architecture by a code unit reallocation effort. In that effort, more code units were deleted, namely those with a zero utilization value. We also gathered data regarding the number of changes against this second deletion set in the time prior to deletion. Then as soon as they were deleted we projected a discounted change rate and we quantified the avoided development effort.

The historical effort for average time spent on each code unit in the time prior to deletion was approximately 3 person-days per year (24 person-hours). Due to planned enhancement projects we could predict a change rate of 70% for the first and second quarter of 2000, and sampling these quarters showed us an average of 6 days spent on each code unit. We discounted this projection with 50% in order to arrive at a fairly conservative estimate. Overall, we chose a conservative 3 days effort per code unit. We use the average but fictitious $1400 as the cost for each day of work. For the average fully loaded developer daily rate paid in North America we refer to [25, 26]. All in all, deleting code units resulted in the saving of 1.9 million U.S. dollar in two years time by avoiding development costs.

Reduced build costs

We assembled historical relevant build data by business line and site. This included the number of builds, the amount of code units per build, the total time to build, the average costs to build, and total build costs of a release. In this way we obtained a sound baseline for further calculations. On the build execution side of this, it meant that there were fewer dependencies, fewer code units, and smaller builds performed. Therefore, the genericity layer reallocation effort markedly decreased build execution costs for the majority of the releases of the system.

Going from phase I to II, we deleted and reallocated some code units (see Table 1). We determined the decrease of the number of code units per build. With that figure we calculated the cost reductions for the first reallocation effort. After phase III we established the base-line federated architecture, where a large movement of code units was realized from the global layer to regional layers, which halved the size of the global layer (see Table 1). Additionally we deleted more zero utilization code units. We calculated the build cost per code unit with the following formula:

$$\text{build cost per unit} = \frac{\text{total effort} \cdot \text{rate} \cdot \text{total \# of builds}}{\text{total \# of code units}}$$

The total build effort can be split into three parts: actual execution effort by release management (76%), preparation and review by source code management (15%), and the efforts by development to correct build failures, e.g., compiler errors (9%). The percentages are specific for this system, we expect them to vary for other systems, especially when the number of installation sites differ from our case.

We determined the new amount of code units for the total system, and the amount of units in each genericity layer. We established certain ratios between the genericity layers to compute the impact of a build given a change in one of the layers. We computed the ratios as follows.

- A global build affects all sites, so the build weight factor of a global code unit that was modified and had to be build at all sites should equal the total number of sites. In our case, this factor was more than 15.
- The build weight factor of a regional code unit only affects one line of business, and therefore a limited number of sites. In our case the variation in number of affected sites for all business lines was not too large so we used the average number of sites that are impacted by a regional build, and took that number as weight factor. When the average has a wide variation, we recommend to use for all business lines separate weights. In our case, this weight was 5.
- The build weight factor of a local code unit only affects a single site, so the factor to take equals 1.

With these weights we calculated costs that would have been spent for updates that were necessary had we not applied the reallocation. Then, we measured the actual build costs after we put all the code units into the correct genericity layer. In our case, the reallocation effort reduced the amount of build costs with 0.6 million US dollar in two years.

Reduced build failures

Apart from successful builds, there are also build failures. Not doing unnecessary builds in the first place leads by definition to builds that do not fail. But also due to the size limitation of the necessary builds the number of failures dropped, and the costs of repairing failed builds dropped as well. We gathered historical data on the total number of builds at each site, and the total number of failed builds prior to implementing the federated architecture. We calculated the build repair costs based upon release management rebuild costs and development repair costs. Then we projected savings in the future and later on measured the actual change in build fails. We measured a cost reduction of 0.3 million US dollars in two years.

Reduced development test costs

Since we deleted a number of code units, it was no longer necessary to test them. Also streamlining the system into a product line eased the testing effort, since the multi-version aspect of the system cascaded down into the testing as well. We gathered historical data on the test efforts, for each business line and site. We calculated the average test costs per quarter prior to our code unit reallocation effort. We normalized the effort on number of issues tested, since if you test more issues the total amount of test costs can go up, but if the cost per tested issue drops, there are still savings. Using this normalized metric, we showed that the costs that we avoided by testing less code units in 2000 and 2001 were about 0.9 million US dollars.

Reduced release digestion costs

There are substantial costs and efforts associated with the digestion of a full release by the user community. The global integration and user acceptance testing effort for releases is enormous when an inordinate number of locally meant but globally implemented changes—irrelevant for most sites—have cascaded into those releases. Since we now intelligently manage the federated architecture, we strongly reduced the effort to digest a release. Users received release contents only when it made sense: where the metrics support having the given code unit in the layer. Regional changes were included for them only where they stood to benefit from these changes. Changes from the core layer cascaded through only when there is high utilization, low volatility, low-to-medium complexity, and low-to-medium specificity—where there is a blend of code characteristics such that the receiving users will benefit if a change is made there.

Moreover the speed to accept a release increased significantly, and the risks of erroneous releases were markedly reduced. Improperly configured source leads to increased errors, more frequent builds and patching of errors, larger builds and more frequent global impact. On the conservative side, a single full release takes 20 person weeks of effort at the five major use-sites, 10 weeks at ten medium sites, and another 5 weeks at five experimental sites. This totals to 225 person weeks, which is 1125 working days, or 5.625 person years (taking 200 working days per year). Not every tester, at every site costs the same, but conservatively they cost a company half the average we used, so $700 US per day, which comes to $787500 US. In this effort user acceptance testing, integration testing, compilation etc. is incorporated. Add another $58800 for the realignment costs, to find the total costs of a single full global release: $846300.

	Full release	LOB release	Site release	Totals
Pre	4725000	0	500000	5225000
Post	1692600	1260000	250000	3085000
Reduction	3032400	-1260000	250000	2140000

Table 7: Pre and post realignment total release cost breakdown.

It is important to realize that if you do *not* realign code units to their appropriate genericity layers, there will be many more unnecessary full releases, at a conservative cost of $787500 US each. Since changes that are local from the business point of view can easily be global from the code unit view, a local business change can lead to a global full release, whereas a local release would have sufficed. A local release at a major site conservatively costs $70000, at a medium site this amounts to $35000, and a small site costs $17500. So, even the most expensive single-site release is a factor 10 cheaper than a global release. From our measurements prior to realignment we inferred that the savings of avoiding unnecessary releases are greater than that for reduced build costs due to smaller build size by a factor of at least 5. This estimate is not easily turned into an exact dollar amount, since there are clearly relativistic effects at stake: intelligent code duplication does increase costs on the part of the development area, but it simultaneously lowers overall cost and effort for the company, because testing and building costs at all the sites outweigh that of development. Moreover, they lower business impact, increase focus, and simplify delivery, leading to higher quality for the company. We summarized annual release digestion costs prior to realignment (1999) and after realignment in Table 7. Prior to realignment, there were at least 6 full global releases, and this was reduced to 2 full global releases. The 6 full releases costs $787500 each, and the post realignment releases cost $846300 each. Prior to realignment, there were no releases to single lines of business, and this increases to 6 releases per year after realignment. This is a cost increase of 1.26 million dollar. There used to be many site releases, at a fairly low cost, and this dropped with half the amount after realignment. In total the annual savings are 2.14 million dollar.

Release indigestion Prior to realignment some sites could not always keep pace with the frequent release rate. This often led to sites being unable to digest all releases. This effect is incorporated in the release costs prior to realignment (Table 7). The problem of what we call *release indigestion* is that it induces more software mitosis than is desirable. Therefore, maintenance at such sites becomes more expensive. After migrating to a product line we established an acceptable frequency of full releases, and only additional releases to a site when their business demanded this. This cured release indigestion, and lowered operational costs. The danger of release indigestion is that sites suffering from it will sometimes miss updates that are crucial to prevent failure of the system. Business failure of a GTSS can incur considerable losses. To give you an idea of the cost of failure: the industry average cost of failure for brokerage operations is $6.45 million dollar per hour [27]. Another report mentioned that a U.S. Based online brokerage firm reported it lost over $100000 per minute when its trading system went down [28]. By curing release indigestion we further reduce potential failure risks, but it is not that easy to put a price tag on such aspects.

Indirect and inferred benefits

Apart from the direct benefits, there are also indirect and inferred benefits. Those numbers are much harder to calculate since observation periods of larger duration are necessary to capture them and because it is sometimes hard to attribute a certain cause to an indirect benefit. We mention a few such benefits:

- We gained a better understanding of the entire system, leading to improved system maintenance, more control, and insight in implementation characteristics and the use of system diagnosis allowing further study and refinements of benefit measurements.
- We decreased the reliance on individual expert knowledge due to a reduction in the physical number of system components and the resulting system complexities.
- The federated architecture is a natural layering to assign staff with varying expertise of the system: the more knowledgeable the developers, the more generic their work can be. Also these experts can spot emerging code units in specific layers and plan to re-layer them in advance.

- A federated architecture allows reuse if that is appropriate and adjusts reuse to the optimal level, but it does not force-fit reuse in all cases irrespective of business need.
- The development group observed a shorter problem diagnosis phase. They attributed this to the more transparent code unit distribution: it seemed more easy to quickly localize the problematic code units. They also felt that there were fewer developer errors due to better organization of the code units.
- The maintenance group felt that due to the product line migration there was no need anymore to remember all the different versions and code variations. This reduced complexity aided in less errors during maintenance.
- The releases to the business are less frequent, and less complex. And when there is a release it is either the semi-annual update, or a release the site asked for. This improved customer satisfaction at the use-sites.

Conclusions

Before we migrated to a software product line, we were in the situation that global components contained many local code units introducing many global updates that should have had a restricted impact. Moreover, regional projects were either done locally or globally, leading to a fat kernel and bloated clients at the use-sites. This led to software mitosis and cost inflating. Now that we migrated to a product line and deploy it we are in the following situation:

- Global code units and components are only altered when the impact of a proposed modification is indeed system wide.
- Regional projects are understood to be important, are routinely sponsored, and regional code units are used to align changes to business line practices. Also when local changes emerge at various sites, the local code units are realigned to more generic layers.
- Local code units only contain site-specific requirements.
- We established direct annual cost savings of approximately 4 million dollar per year. This saving is about 3–7% of the total annual budget in our case, but can be higher if the number of use-sites is higher, and lower if the number of use-sites is lower.
- The indirect benefits are a flexible system that allows for rapid directed growth, and a federated architecture that is a competitive edge in mergers and acquisitions.

Acknowledgements Thanks to Henk Obbink, Rob van Ommering (both Philips Research), Dennis Smith, Christoph Stoermer (both SEI), and the anonymous reviewers for discussions, comments, improvements, and contributions.

References

[1] M.L. Fisher. What is the Right Supply Chain for Your Product? *Harvard Business Review*, pages 105–116, March–April 1997.

[2] J. Bosch. *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[3] P. Clements and L.M. Northrop. *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2002.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] P.A. Strassmann. The Policies and Realities of CIM – Lessons Learned. In *Proceedings of the 4th Armed Forces Communications and Electronics Association Conference*, pages 1–19. AFCEA, Fairfax, VA, USA, 1993.

[6] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[7] R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[8] H.M. Sneed and C. Verhoef. Business process reengineering. In J.J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 83–95. Wiley Inc., 2 edition, 2001.

[9] L.M.G. Feijs, R.L. Krikhaar, and R.C. van Ommering. A relational approach to Software Architecture Analysis. *Software Practice & Experience*, 28(4):371–400, April 1998.

[10] R.L. Krikhaar. Reverse Architecting Approach for Complex Systems. In M.J. Harrold and G. Visaggio, editors, *Proceedings of the International Conference on Software Maintenance*, pages 4–11. IEEE Computer Society, 1997.

[11] R. Krikhaar, A. Postma, M.P.A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 371–380. IEEE Computer Society Press, 1999.

[12] R. Kazman, L. O'Brien, and C. Verhoef. Architecture reconstruction guidelines. Technical Report CMU/SEI-2001-TR-026, Software Engineering Institute, 2001.

[13] L. O'Brien, C. Stoermer, and C. Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, 2002.

[14] A.A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November/December 2000. Available at http://http://www.cs.vu.nl/˜x/cnv/s6.pdf.

[15] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control Flow Normalization for COBOL/CICS Legacy Systems. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://www.cs.vu.nl/˜x/cfn/cfn.html.

[16] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. *Science of Computer Programming*, 45(2–3):193–243, 2002.

[17] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.

[18] V.R. Basili. Software modeling and measurement: The Goal/Question/Metric paradigm. Technical Report CS-TR-2956, Department of Computer Science, University of Maryland, 1992.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[20] J. Rumbaugh, M.H. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[21] G. Booch. *Object-oriented Analysis and Design – with Applications*. Object Technology Series. Addison-Wesley, 1994.

[22] M.H. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[23] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[24] F.P. Brooks Jr. *The Mythical Man-Month – Essays on Software Engineering*. Addison-Wesley, 1995. Anniversary Edition.

[25] C. Jones. *Software Assessments, Benchmarks, and Best Practices*. Information Technology Series. Addison-Wesley, 2000.

[26] The Bureau of Labor Statistics. Computer Programmers. In *Occupational Outlook Handbook, 2002-03 Edition*, pages 166–169. Bureau of Labor Statistics, Chicago, USA, 2002.

[27] Fibre Channel Industry Association. Business continuity when disaster strikes, 2000. Available via: `www.fibrechannel.org/solutions/Continuity/Continuity.pdf`.

[28] The Standish Group. Five “T’s” of Database Availability, 1999. Retrievable via: `www.sybase.com/content/1020238/StandishFiveTsHA.pdf`.