Experience with the Architectural Design of a Modest Product Family

Robert W. Schwanke
Siemens Corporate Research Inc.
755 College Road East
Princeton, NJ 08540-6632
robert.schwanke@siemens.com

Robyn R. Lutz
Department of Computer Science
Iowa State University
Ames, IA 50011-1041
and Jet Propulsion Laboratory
rlutz@cs.iastate.edu

Summary

Many product families are modest in the sense that they consist of a sequence of incremental products with, at any point in time, only a few distinct products available and minimal variations among the products. Such product families, nevertheless, are often large, complex systems, widely deployed, and possessing stringent safety and performance requirements. This paper describes a case study that tends to confirm the value of using a product-line approach for the architectural design of a modest product family. The paper describes the process, design alternatives, and lessons learned, both positive and negative, from the architectural design of one such family of medical image analysis products. Realized benefits included identifying previously unrecognized common behavior and sets of features that were likely to change together, aligning the architecture with specific market needs and with the organization, and reducing unplanned dependencies. Most interesting were the unanticipated benefits, including decoupling the product-family architecture from the order of implementation of features, and using the product-family architecture as a "guiding star" with subsequent releases moving toward, rather than away from, the planned architecture.

Key Words: Software Product Family, Software Architecture, Software Product Line, Medical Image Analysis, Modest Product Family, Product Line Architecture (PLA)

Introduction

Software product families vary widely in the number of members and the variety of options. Many product families are "high-end" in that they contain a large number of fairly diverse systems, often with automatically generated instantiations and fine-grained reuse of assets. Other product families are what we term "low-end" in that they contain fewer systems and less variation in options, often with only minimal customization and configuration for individual customer needs. It is these more modest product families that the work reported here investigated.

Since the greatest benefit is presumably to be obtained from the most reuse, attention has tended to focus on the product families that offer the greatest opportunity for reuse, namely those with many family members providing many different combinations of features. Examples of such product families include command-and-reporting software [1], servers [2], and diesel-engine software [3]. In some of these high-end product families there are domain-specific languages to describe the product family, allowing the software for each product in the family to be automatically generated from the decision model [4].

Less attention has been directed at modest product families, i.e., those with fewer instantiations and fewer opportunities for customization, despite the fact that they are common in industry. Reasons for the wide-spread use of modest product families include that many organizations do not have an infrastructure or culture that supports reuse, that tool support for automated generation of a new product from an existing set of assets is

still not widely used, and that the expected market life of many systems is seen as too short (often two to five years) to make the broad reuse offered by a product-family approach beneficial.

The major contribution of this paper is to suggest that product-family architectural techniques are effective for even modest product families based on our experience with the architectural design of a family of medical image analysis products. This product family used only run-time mechanisms to control product variations and its product-family architecture was nearly identical to the architecture of each product in the family. The paper presents the architectural techniques used, the key design and planning strategies chosen, the architecture and associated project plan produced, and the impact that the techniques had on the success of the project. (Only the name of the product family, "Viscard," is fictitious. Everything else in this report is factual and undisguised.)

Product Line Practices

The first generation of Viscard, developed prior to this study, had not been a product family. It consisted of four separate products developed in independent projects with minimal shared assets and otherwise only copy-and-modify reuse. The project described here designed and built the second generation of Viscard systems as a product family. The family consisted of new generations of the original four products, plus perhaps four to eight additional, expected products.

Viscard is a set of software systems for viewing and quantifying radiological images of cardiovascular organs, especially the heart. The systems share a common set of general-purpose features for viewing and quantification, while each system has additional features specialized to a particular kind of image or a particular quantification technique. Viscard's market segment consists of hospitals, clinics, and private medical practices that diagnose cardiovascular diseases. The systems share a common architecture, application framework, and look-and-feel, and are developed concurrently under what has been, until recently, a single project.

The Viscard project was only chartered with producing a set of products in an effective manner. In fact, initially only the architect was conversant with product-family concepts. Nonetheless, many product-family practices were actually used. These practices were mainly software engineering and technical management practices, and not organizational management practices. The rest of this section describes Viscard in terms of the SEI framework for product-line practices.

The Software Engineering Institute defines a software product line as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [5]. The Viscard product family was thus a product line as well as a product family.

The architecture requirements were documented primarily as dimensions of expected variation, whether the variation would be concurrent, because of differences between products, or sequential, because of anticipated future trends. The architecture was evaluated both for its suitability to meet immediate, known requirements and to be maintainable in the face of plausible future variation.

Viscard employed a modest notion of "component" to organize the common and product-specific code. The common application framework has approximately eight hot spots where a product-specific component could be plugged. Each hot spot provided a generic façade class [6] that defined the interfaces the component was required to provide. The façade class also identified the interfaces that the component was permitted to use. Some of these interfaces provided infrastructure, whereas others provided classes that implemented common features. Each product-specific component was developed as a separate Microsoft Developer Studio project, producing a separate DLL. Separate compilation and linking was then used to assure that only permitted interface dependencies occur.

Most of the COTS decisions affecting Viscard had already been made when the project started. Viscard was chartered to run on the proprietary *syngo*®[1] execution platform, which is used by most Siemens medical imaging products. This platform provided a great deal of useful infrastructure, a wealth of medical imaging code assets, and the configuration mechanisms needed to embed Viscard in larger products and bind it to the computer configurations on which these products would run. *syngo* also mandated a common look-and-feel and a common COTS software development product (MS Developer Studio). The main choices left to Viscard were the extent to which it used the *syngo* imaging code assets vs. building its own.

The project mined some existing assets from the first generation of Viscard products: some key algorithms, some data structure designs, many functional requirements, and many user interface design elements. In particular, the previous products were used as the basis for domain analysis and product-family scoping, yielding a single unified concept of a medical image post-processing "toolkit". This concept guided the organization of the common application framework. Three other, core assets were heavily influenced by make/buy/mine/commission analyses: a generic View framework, a component that plugged into that framework, and a building block for viewing moving images.

The development process for the new generation of Viscard systems resembled the development of a single product more than the development of a product family for several reasons. The core assets and the first two products were developed concurrently in one project, assigning each developer some common assets and related, discipline-specific code. This made it convenient to have a single software build process that produced the core assets and the individual products. A continuous integration strategy brought all the code together frequently, detecting and correcting integration problems early in the process. The entire output of this build process was shipped to all customers. Installation tools then configured the products to the needs of the individual customer.

Sharing the project and the production process streamlined many technical management practices. The source code versions were managed as a single, multi-site code base. The core assets and per-product components were clearly separated in different MS Developer Studio project directories. All of the per-customer configuration information was handled by installation scripts and load-time parameters. In the project plan, the core assets did not have separate tasks or delivery dates, because they were developed

[1] *syngo* is a registered trademark of Siemens AG.

conjointly with the first two products. Technical risk analysis combined product-family and single-product risks in a single, managed list.

Related Work

The architectural analysis and design of product lines has been extensively investigated in the last few years [7, 2, 8, 9, 10, 11]. Most of this literature describes product families in which a large number of systems, customized to specific market needs, are supported at the same time. Little attention has been given to investigating the appropriateness of product-line architectural methods for more modest product families, where there is a sequential release of a few, tightly controlled products with few variations among them. We describe in this section the work to date on the architectural analysis of such systems.

Among the exceptions to a focus on large-scale reuse [12] is the work by Clements and Northrop [5] and Knauber et al. [13]. Both describe the existence of small and medium-sized companies with lightweight product line processes and the important role of architectural analysis in guiding development.

Thiel and Perruzi describe a product family that is modest largely because it is new. They found that a product-line architectural approach worked well for the new envisioned system [14]. They describe how in a very competitive environment where innovative products are essential, a new product may often be undertaken without benefit of reference product knowledge and experience. For their product (integrated dashboards for vehicles), they found that the systematic analysis of possible customer-driven variations helped define the domain. In addition, they found that the architectural analysis (using ATAM, the Architecture Tradeoff Analysis Method, and an early version of ABD, the Architecture Based Design Method [15, 16]), helped create templates for subsystems and components. Since they are describing the early development phases of a new product family rather than the results of an on-going product family (as with Viscard), some of their conclusions are speculative. The results presented here tend to confirm their speculation that a product-familyarchitectural approach yields benefits even for the initial release of a product.

Similarly, Voget and Becker [17 found that a product-family approach provided methods that overcame some risks of designing systems in an immature domain. Where future requirements are vague or unknown, what they call "lightweight domain engineering" promoted early analysis of risks and constraints. However, their focus was on identifying reusable assets (components) rather than on a reusable architecture. The Viscard product family also had high anticipated change over time in response to market forces and both predictable and unpredictable technological advances in hardware and algorithms, but found that an architectural focus was key to managing risks.

In contrast with Viscard, both Winjstra [18] describes a product family with a relatively small number of systems but (unlike here) a stable domain and relatively mature market. The product family is modest because it contains a limited number of systems with limited options. The stability of the domain permitted a domain-specific architecture with units that each represented a phase in the workflow (e.g., patient administration, image viewing, etc.).

Pronk described the architecture for another product line in the same domain, currently in its first release [19]. This product line is considerably more ambitious than the modest Viscard product line. Unlike Viscard, there are many concurrent versions of specific products, differing in the acquisition hardware and software and in the image processing functionalities. It defines an application framework for image acquisition systems, with "plug-in" interfaces for each of the components that are expected to be different for different products. Also unlike Viscard, new features needed to be added rapidly in response to market changes without releasing a new platform.

Variation Analysis

Variation analysis for single products normally examines two kinds of variation: uncertainty in the current requirements and changes to those requirements over time. For software product families, there is a third kind: variation between products. These can be viewed as three largely independent dimensions of variation. For example, suppose that two products in a product family require different image processing algorithms. The CPU performance required to process images varies depending on the algorithm used (varying between products), the efficiency of the algorithm (uncertain until the algorithm is implemented and optimized) and the size of the image to be processed (increases over time). Despite the dimensions being largely independent, their effects on the architecture may be combined into a single aggregate effect. In the image processing example, the aggregate effect is that the architecture is required to support cost-effective implementations across a broad range of image processing performance levels. In this section, we use the terms "uncertainty", "change", and "variability" for the three dimensions, and "variation" as a generic term for any or all of the dimensions.

"Commonality" is in one sense the opposite of "variability", referring to the extent to which products have common properties or requirements. However, it goes further than that, because commonalities suggest opportunities for asset sharing among products. For example, if two products require the same annotation capability, they can potentially share the requirement, the specification, the implementation, and the tests associated with that capability. Later on, should their requirements diverge a little, they may still be able to share the other assets, but the assets may have to be reworked to satisfy the two requirements at the same time.

In this section we describe the variations anticipated for the Viscard product family as the architecture was being created. We start by describing the domain of medical image post-processing and the scope of the product family within it, ending with some examples of sharable assets. In the second subsection, we discuss other important factors influencing the architecture, including the organization and the technology available. Later in the paper, some unanticipated variations are described to evaluate how well the architecture handled them.

Domain Analysis and Product -Family Scoping

The goal of domain analysis and product-family scoping for Viscard was to come up with a way of describing the space of potential products to include in the family, described in such a way that the practical commonalities and variabilities would be evident. The

analysis began with four existing first-generation products and knowledge of other products being developed elsewhere, inside and outside the company. It generalized these products to define a *product-oriented domain model*, bounded the *product-family scope* within that domain, scoped the first few second-generation products, and then scoped the *common assets* to be used in those products [12].

Medical radiology imaging technology supports the diagnosis and treatment of diseases that cannot be adequately observed with visible-light imaging or the naked eye. Several different imaging modalities are currently used, including ultrasound, magnetic resonance (MR), computed tomography (CT), X-ray, positron emission technology (PET) and nuclear imaging. These imaging technologies differ in image resolution, number of dimensions, and ability to distinguish different kinds of tissue (e.g. bone, soft tissue, and blood). Two special cases highlight the variety of possibilities: nuclear imaging shows differences in how rapidly tissue absorbs radioactive dyes, and some kinds of MR images show the tissue's velocity (e.g. blood flow) as well as its density.

A typical workflow in a radiology department of a hospital goes through the following main steps. (The italicized words are significant technical terms from the domain):

- A *referring clinician* orders a radiological *exam* to help diagnose a type of disease in some area of the patient's body.
- A *technician* conducts the exam, following an image *acquisition protocol* to acquire raw image data and *reconstruct* a set of viewable images, called a *series*.
- The technician views the images to assure their quality.
- A *radiologist* views the images and *reports findings*, typically by *annotating* the images, *quantifying* (computing measurements from) the images according to a *post-processing protocol*, and dictating the findings for later transcription. The report includes significant images and a link back to the complete series.
- The clinician reviews the report (electronically or from hardcopy), and perhaps the original series, and makes a diagnosis.
- The radiologist and referring clinician may confer about the findings and diagnosis.
- The referring clinician may show the report to the patient while explaining the diagnosis.

Siemens sells most types of medical imaging equipment and software used in the radiology workflow. In the past, most products have been designed for the radiologist and her staff, but new products are beginning to appear that are designed for use by certain kinds of clinicians, such as cardiologists and oncologists. Many of the products are designed for use with a particular imaging modality, but others, especially image archiving and viewing products, are designed to support many imaging modalities.

Schmid's taxonomy of product line scoping methods defines the term "domain" such that domains can have sub-domains, domains can overlap, and a product can address several domains and sub-domains. A product line can be scoped to address, not only the domains of the specific products currently planned, but also the domains that additional products could reasonably address and remain within the product line. In this case, the domains are the modalities, the medical specialties, the acquisition and post-processing protocols, and the steps in the radiology workflow. Within the protocol domains are the sub-domains

representing different viewing, annotation, and quantification capabilities, different organs in the body, and different types of diseases.

The first four Viscard products addressed the following domains: the MR imaging modality, radiologists and cardiologists, cardiovascular organs and diseases, four particular post-processing protocols, and all of the radiology workflow activities except the original order and the image acquisition. Viscard would be deployed in two principal types of hardware configurations: as part of an image acquisition workstation, and as part of an image review workstation. Workstations would vary in the size of display, size of memory, and processing power.

However, the project was also chartered with defining an architecture that would support future post-processing protocols emerging from basic research. When research on a new imaging or diagnostic technique has shown sufficient promise, a company like Siemens partners with a medical research group to build a software package, called a Work-In-Progress (WIP), to begin making the technique practical. The WIP goes through several cycles of experimentation and redesign until it is ready for use in clinical trials. If the trials are successful, the package is rebuilt into a marketable, "clinical" product. A WIP version may continue to evolve after the clinical version is in production, as researchers continue experimenting with new and improved techniques. Thus we see that the particular toolkits to be built, when development will begin, and when they will be ready for the market are all highly uncertain. The functional specifications for any particular toolkit remain volatile throughout the WIP phase and well into development of the clinical product.

To define the product-family scope, the analysis added several domains that were likely to be addressed in future products. Two imaging modalities, CT and X-ray angiography (X-ray with opaque die in the blood vessels), were expected to become practical for cardiovascular diagnosis in the future. (Ultrasound and nuclear medicine were already used for cardiovascular diagnosis but were not included in the product-family scope since they yielded much lower resolution images, and their post-processing tools and techniques were dissimilar to the original four products.) Although all the known MR cardiovascular protocols used only 2D image viewing methods, 3D viewing methods such as multi-planar reconstruction (MPR) and shaded surface display (SSD) were added as potential future sub-domains in the product family. The scoping analysis also added the image acquisition step as a potential future sub-domain: although the Viscard product would not acquire images, it would be used to view and analyze "navigation images" that guide the technologist to the best acquisition protocol parameters. The analysis also took into account that Viscard would be useful for other organs of the body in the future.

Commonalities

From the four first-generation Viscard products and knowledge of other products in the product-family domain, we distilled the following common concept of a post-processing protocol toolkit that formed the core of the product-family architecture.

A complete protocol for a radiology study is a sequence of steps that a technician or doctor follows to acquire the images and analyze them. It consists of two main parts: the acquisition protocol and the post-processing protocol. The acquisition protocol specifies

how to prepare and position the patient, select the right control parameters for the equipment, etc. The post-processing protocol specifies what kinds of acquired images can be used, how to find and mark the regions of interest in the images, how to take measurements from the images, such as the thickness of a heart wall or the volume of blood flowing past a certain point in an artery. In addition to the steps explicitly specified in the protocol, the person analyzing the images performs other steps of her own choosing, using general-purpose viewing, annotating, and measurement capabilities of the software.

Figure 1 is an activity diagram that specifies the steps of a blood flow analysis post-processing protocol. The arrows between steps indicate data that is transferred from the output of one step to the input of another. Notice that there is a cycle in the graph, feeding some of the results from one step back to an earlier step. This is not an infinite loop, but is a form of feedback to check whether the earlier step was done accurately enough, or whether it needs to be redone.

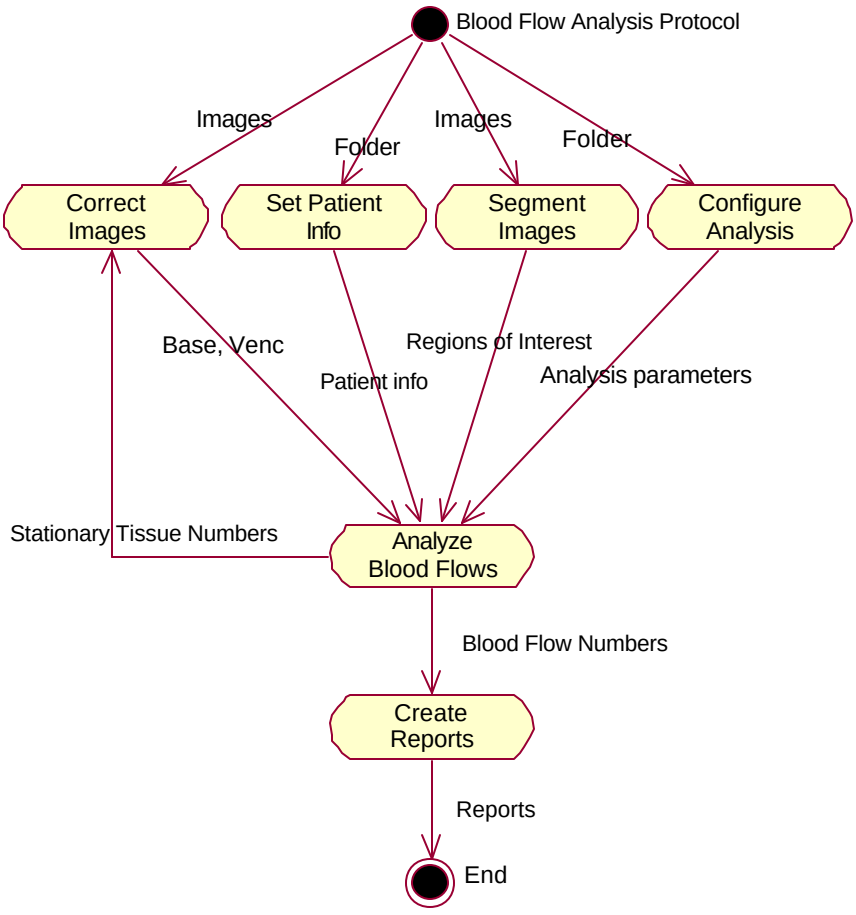

Figure 1 Directed Graph of Steps of a Blood Flow Analysis Protocol

A protocol toolkit consists of general-purpose post-processing tools, special-purpose tools, and zero or more specific protocols to follow to obtain certain kinds of quantified results. The general-purpose tools are mainly for viewing (e.g. pan, zoom, brightness, contrast, and "movie" or *cine*) and annotation (geometric shapes, drawing contours along the boundaries of organs or tumors, etc.) The special-purpose tools typically relate to the type of images being processed. For example, MR "flow" images encode the speed and direction in which the blood is flowing at each point in the region of interest. A viewing tool for flow images color-codes the velocity of blood flow along one axis, but the user must select the axis and its zero point before the color-coding can be done. A protocol links the tools together in a partially-ordered set of processing steps, so that the outputs of some steps are inputs to other steps. For example, to compute how effectively a heart is pumping blood, the user must first identify the images representing the systolic and diastolic volumes of the heart, then mark the inner wall of the left ventricle on each selected image. After that, an algorithm computes the area enclosed by each contour and uses it to determine the systolic and diastolic volumes and the ratio between them.

From this description it is easy to identify some of the common assets that could be shared among protocol toolkits: a viewing and annotation framework, with an organizer that keeps the images arranged in space and time; general-purpose viewing and annotation tools, and libraries of building blocks for the special purpose tools; and a protocol framework that manages the partial ordering of protocol steps and connects their inputs and outputs together.

Figure 2 shows a cross-section image of a heart, with contours marking the inner and outer walls of the left ventricle.

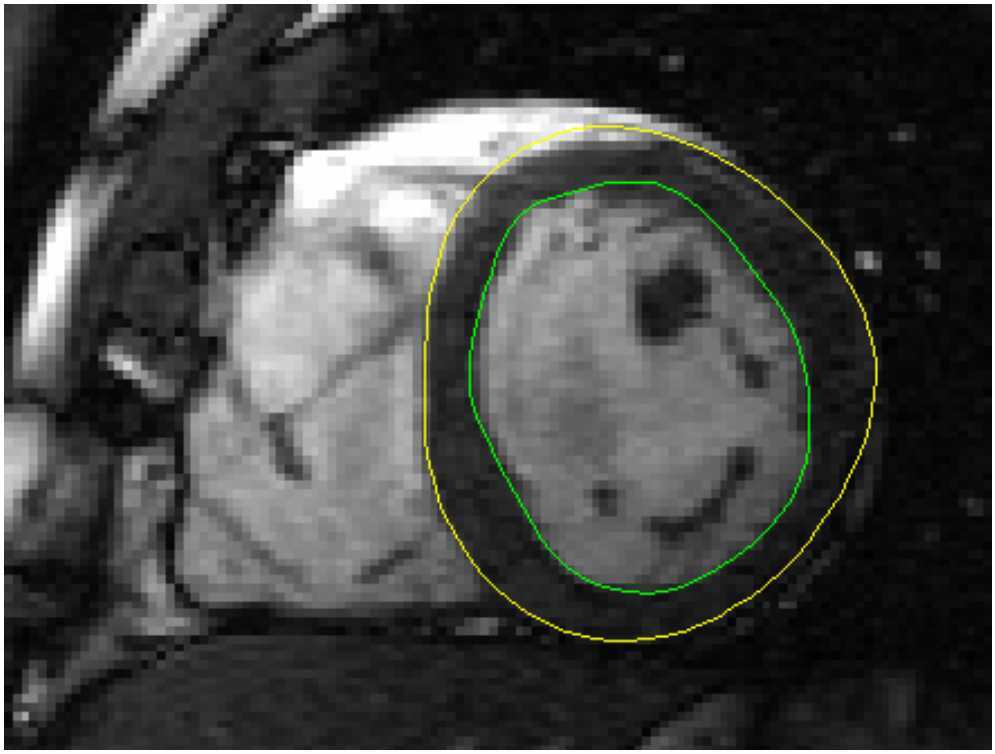

Figure 2 Cross section of a heart, with contours marking left ventricle wall.

Expected Product Variations

In addition to the product-family variabilities already discussed, the Viscard project anticipated that the following variations within individual products could affect the architecture. Some of these differences involve properties of the product itself, but many instead involve the development organization or the technologies available to develop the product.

GUI design details. The GUI designs would need to be revised and refined iteratively, based on user feedback.

Expert vs. guided usage. Viscard should accommodate both expert users and new or infrequent users. Expert users should be allowed as much flexibility as possible, whereas new or infrequent users should be carefully guided through the basic steps of the standard analysis protocols.

External interfaces. Some of the principles of interaction between Viscard and the MRI system's other components had not been defined at the time of the initial Viscard design. These included a startup-on-demand protocol and a data interface library for accessing proprietary attributes of data series.

Synchronous vs. asynchronous commands. Although most commands were executed synchronously (before accepting another command from the user), some commands might need to be executed in the background, while the user was doing other things.

Image Interpolation. First-generation Viscard products had used only integers as zoom factors when scaling images. Using more sophisticated interpolation techniques would increase usability, but might affect edge sharpness and the placement and interpretation of contours. Therefore, interpolation techniques had to be modifiable late in the project.

PC performance. Although off-the-shelf PC performance was expected to improve considerably over the life of the product, good performance on low-end PC's would increase market share, for example for home use by on-call radiologists.

Image analysis techniques. New image analysis techniques, such as constrained-model deformation, might be added to the product later.

External data formats. Image analysis tools were required to handle old data, since some image archives had been around for many years. The Viscard system thus interfaced with e legacy formats, DICOM data files, and *syngo* databases. It would also be called upon to handle data from multiple generations of the acquisition software.

Display properties. Size, aspect ratio, color vs. grey scale, number of monitors, and pixels per monitor all affect the layout of a GUI. The *syngo* GUI standard called for a 1280x1024 color monitor as the "main" console, initially, but some review stations were already known to have as many as eight monitors, and it was known that both larger and smaller monitors would eventually need to be supported.

In summary, the following variabilities in the case-study product family appear to be sufficiently generalizable to other product-family domains to merit discussion of their architectural implications below: ***Many analysis protocols, WIP(Work in Progress) vs. Clinical versions, Acquisition protocols, Product configurations, Quantification methods,*** and ***Image and result views.***

Other Influencing Factors

A good architecture addresses not only properties of the product line itself, but also influences from the organization that will produce, sell, and maintain the product, and influences from the technologies available to develop the product [20, 9]. Many of these influencing factors changed over time. Here we describe some of technical and organizational factors that influenced the Viscard architecture.

Development tools. Relying on an elaborate development environment like Microsoft Developer Studio and other sophisticated tools like Rational Rose and ClearCase MultiSite made the project vulnerable to schedule delays when switching to new versions and to incompatibilities between versions of different tools.

Early syngo adopters. The Viscard project was among the first wave of *syngo* platform adopters, and was the only early adopter to use a particular *syngo* image viewing API. This made it vulnerable to the same kinds of costs and delays encountered when developing on top of beta releases of new products.

Key personnel in high demand. Because of the hot job market of 1998-99, the project was at especially high risk of losing key personnel to other projects or employers. It was also difficult to acquire new personnel. However, these are slightly different issues, since losing key personnel can kill a project, whereas failing to hire new ones may only make it late.

External schedule dependencies. The project delivery schedule was determined by the delivery schedule of the modality's development project, which in turn was determined in part by the *syngo* delivery schedule. Both of these sets of external dates were unstable. This forced the Viscard project to adopt aggressive delivery dates that were likely to be relaxed later, but by an unpredictable amount.

Safety. Hazard analysis, reused from the previous generation of the product family, identified two safety-critical aspects of the system: producing correct computations and associating them with the correct patient. (These two aspects of the system were carefully encapsulated in the architecture to contain redesign, testing and certification cost, and did not otherwise significantly affect the architecture.)

Pressure for late changes. Developers of the previous product generation had found it difficult to resist pressure to change the product functionality late in the development cycle. These late requests especially involved GUI and numerical formula changes in response to initial feedback from using the work-in-progress versions of the product.

Multi-site development. The decision to distribute the project across multiple sites was already made when the project began.

Limited object-oriented development experience. Many of the then-current and potential team members had not developed object-oriented software before.

syngo technologies. The *syngo* framework provided several novel technologies, and would evolve significantly over its lifetime. Its future performance capabilities were uncertain. These novel technologies included object-oriented databases, image-processing libraries, novel error logging and tracing capabilities, DICOM[2]

[2] DICOM is a healthcare industry standard for digital image communication in medicine.

communication, asynchronous remote method invocation, dynamic reconfiguration, and componentware tools, techniques, and standards.

New technology overload. The project was adopting a large number of new technologies, making it difficult to train new staff and to anticipate the interactions between technologies. New technologies included Windows, C++, Microsoft Developer Studio, Microsoft Foundation Cla sses, OLE Controls, Rational Rose, WinRunner, ClearCase MultiSite, and, of course, the *syngo* technologies.

From the variation analysis [21] and other influencing factors described above, a small number of key issues emerged to drive the architecture and the project plans. An architectural issue normally represents a conflict between two or more influencing factors. Each important issue tends to involve mixtures of product, organizational, and technological factors, and can be addressed by a combination of technical and managerial solution strategies. Normally, the architect finds that there are many influencing factors defining each key issue, that some factors contribute to more than one key issue, and that some solution strategies address more than one key issue. Finding a small number of solution strategies that, together, adequately address the key issues goes a long way toward a successful project.

We describe three key issues from the Viscard project: unstable protocol requirements, unstable planning constraints, and image processing performance. For each of these issues, we discuss the relevant influencing factors, the overall solution approach chosen, and the specific strategies used.

Key Issues and Strategies

Unstable Protocol Requirements

Variation analysis showed that the requirements for Viscard protocol toolkits were unstable, due to several factors. ***Image acquisition techniques*** were being invented or improved regularly by the imaging hardware designers. Algorithms specialists were inventing ***new quantification*** and ***image viewing techniques***. ***Many*** diagnostic ***protocols*** had not yet even been defined, but would emerge from basic medical research and be implemented and incrementally improved as ***WIPs*** before becoming ***clinical products***. The GUI had a difficult, new requirement to provide "guidance" for new and casual users, without hampering expert users. ***GUI design details*** would continue to evolve as the clinical product was introduced to potential users. Inevitably, customers would apply ***pressure for last-minute changes***. Furthermore, this was the first time that the products would be sold directly to cardiologists, and not just radiologists, which called for packaging the technology into products in new combinations.

Thus, the number of toolkits to be built and the specific requirements for each toolkit would remain uncertain throughout the initial architectural design phase of the project. During the WIP phase of each toolkit the detailed requirements for that toolkit would be unstable, with consequent risks to cost, schedule, and quality, making unstable protocol requirements a key concern for the project.

The general technical approach chosen to minimize their impact was to design for requirements change; specifically, to reflect the product-oriented domain model in the architecture, to encapsulate different types of code affected by different kinds of change, and to use incremental design and development methods that help stabilize requirements. In addition, the project plan was organized to allow for certain kinds of anticipated change.

The most important technical strategy was to organize Viscard as a ***product family of protocol toolkits***, rather than as loosely-related products. This gave the products a shared architecture that reflected the domain structure as well as shared design and code assets. Shared assets were relatively unaffected by the changing implementations of individual tookits, and, conversely, changes encapsulated in shared assets had less impact on the design or implementation of toolkit-specific subsystems.

The first structural principle in Viscard was to organize it as an ***application framework for protocol toolkits***. The framework implemented a "main program" with all the generic features of a protocol toolkit, including: interfacing to the *syngo* platform; beginning and ending a session; loading, unloading, and displaying images; providing menus and tool cards for the tools; sequencing through the steps of a protocol; and, saving and printing the diagnostic findings. The framework had ***hot spot interfaces*** [22, 23] for plugging in toolkit-specific functionality. Each such interface was specified by a base class with the expected methods and properties, from which each toolkit derived a subclass with protocol-specific implementations and extensions of the methods and properties. The hot spot interfaces decoupled the common code of the framework from the toolkit-specific code.

The application framework was further decomposed according to the **Model-View-Controller (MVC) pattern** [6], which decouples these three types of code by placing them in separate subsystems. Each subsystem had one or more hot spot interfaces for the toolkit-specific part of the model, view, or controller, respectively. In addition, each subsystem had a library of ***reusable toolkit building blocks***, so that different protocols could be built out of common parts. Some of the common building blocks were designed from the start to be reusable, while others emerged later when opportunities for reuse were discovered.

Having somewhat isolated the GUI code with the MVC pattern, the project tackled the GUI stability problem by doing ***rapid GUI prototyping with frequent customer feedback.*** Starting with a mockup (implemented as a PowerPoint® presentation with hyperlinked pages), the project showed the proposed GUI design to a variety of stakeholders, especially radiologists who would be early adopters for feedback early in the development process. Parts of the mockup were incrementally replaced with screen shots and code. Writing and reviewing of the GUI details of the Functional Specification was then deferred until the prototype stabilized. This approach supported a novel user interaction concept, i.e., ***two user modes with crossover.*** This consisted of a general toolkit mode for expert users and a "wizard" mode for inexperienced users, with a mechanism for switching modes without losing any of the user's work.

This combination of architectural decisions also implemented two other solution strategies. It ***localized product variabilities*** by wrapping them in hot spot interfaces, and

it made it possible to ***localize new technology dependencies***, by encapsulating dependencies on APIs that represent new technology, and, where possible, limiting the use of new programming support tools to the subsystems where they really made a difference.

Module View

The technical strategies and decisions described above can be seen in the module view of Viscard's architecture. This view was organized into five major subsystems, as shown in Figure 3: a component layer, which embedded the application in the *syngo* component framework; a session layer, which supervised the overall handling of each set of images; and a Model subsystem, a View subsystem, and a Controller subsystem, which realized the Model-View-Controller pattern. Most of these subsystems were further broken down into a Front End (FE) part and a Back End (BE) part, connected by a communication wrapper (Comm), which hid the details of marshalling, dispatching, and un-marshalling the communication.

Each protocol toolkit was realized by deriving protocol-specific subclasses from various "hotspot" base classes. Its controller and model were derived from the base controller and base model classes shown in Figure 3. There was only one view manager, but each toolkit configured it to display both generic and toolkit-specific views. The latter were realized by derivation from generic view classes.

Each tool in a protocol toolkit potentially entailed controls, such as menu items and dialog boxes, by which the user invoked the tool. Each tool also had views, which allowed the user to see the data before and after operations, and in some cases to directly manipulate graphical and image data. Thirdly, each tool had model methods that actually altered the data according to the semantics of the tool. The controls were collected in the Controller subsystem, the views in the View subsystem, and the model methods in the Model subsystem.

This structure facilitated iterative GUI development by isolating a protocol's controller and view front-ends from the back-end code, and by making libraries of reusable code assets available in the BaseControllerFE and ViewManagerFE subsystems.

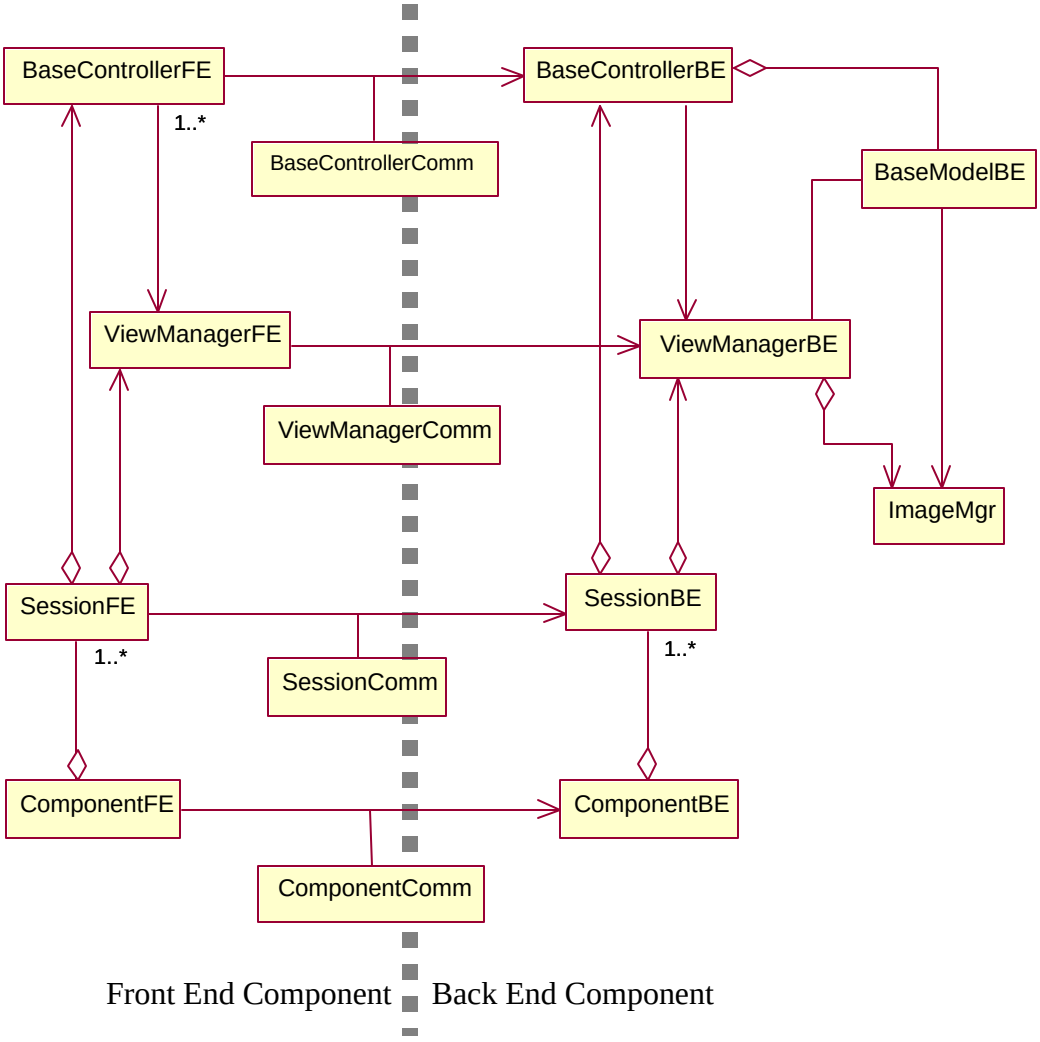

Figure 3. Module view, with assignment to execution components

Unstable Planning Constraints

Many of the factors influencing the Viscard project plan were significantly unstable. Therefore, it was expected that the plan itself would likely undergo frequent and perhaps even dramatic changes. The challenge was to make the plan flexible enough to be changed without disrupting the ongoing work or losing control of cost, schedule, function, or quality.

Although the protocol toolkit requirements were uncertain, the project did not expect big changes once a toolkit entered its clinical implementation phase, unless an important customer exerted ***pressure for late changes***. More difficulties arose from unstable ***external schedule dependencies***, because Viscard was being embedded in a larger product, with its own development schedule, and both were ***early syngo adopters***, subject to the uncertain function, schedule, and quality of the *syngo* beta releases. This forced the Viscard project to adopt aggressive delivery dates that were likely to be relaxed later, but by an unpredictable amount. Because of the hot job market at the time, the project was

vulnerable to the sudden loss of ***key personnel***. The development team would ***acquire new personnel*** by hiring, which made the headcount growth curve dependent on interview outcomes and available-to-start dates. Viscard also depended on several ***external interfaces*** that were ***undefined*** at the project start. Several factors were unstable due to lack of experience: many of the team had ***limited O-O development experience***, the organization was trying serious ***multi-site development*** for the first time; and, the project was adopting a large number of ***new technologies***.

The chosen approach to achieving a flexible plan was ***architecture-centered software project planning*** [9]. The project planning used the module view of the architecture as the basis for breaking the work down into tasks and for bottom-up effort estimation. This facilitated monitoring effort-to-complete, which, when compared to the remaining effort budgeted to the milestone, provided early-warning signals of problems.

This approach included an ***incremental build plan***, in which a skeletal subset of the system functionality was implemented, integrated, and debugged first, followed by the addition of functionality in frequent, planned increments. This approach helped validate the architecture and identify integration problems early.

Product-family development initially used a ***single project, multi-product*** approach, with the plan being to design and implement the initial set of core code assets and the first two product toolkits and release them together. Subsequent releases added new toolkits as well as improving the ones previously released. Later, as the number of toolkits grew and the common assets stabilized, separate projects were planned to develop new WIP toolkits.

Exploratory common asset factoring was feasible whenever the opportunity arose. The same designer simultaneously designed the corresponding parts of two protocols and their common assets. For example, the same person designed the base controller and the first two toolkit-specific controllers. This enabled the designer to decide which code should be shared and which code was toolkit-specific, and to rapidly implement this decision.

Having adopted these basic decisions, many of the strategies used to define the architecture had direct, positive impact on the project plan as well. Perhaps the greatest benefit was that the product-family structure was visible in the incremental build plan as the common assets and toolkit-specific functions were allocated to different builds. The Model-View-Controller pattern showed up as separate tasks for building the model, view, and controller subsystems of the framework and the two products. Each module or subsystem that localized a type of product variability, or a type of technology dependency, or some other important influence, thus showed up as a work package in the plan.

The architecture description provided a sound basis for a modular project plan. The first detailed project plan and cost estimate, at the end of the initial architecture phase of the project, was based on a fairly detailed module decomposition tree. It contained two internal builds for each of the first two external milestones, emphasizing the protocol framework and the first protocol toolkit in the first two builds, and the second toolkit and more thorough testing in the third and fourth builds. The project documented the

assumptions on which the estimates were based and fed those assumptions into the risk management process.

Both the product-family architecture and the derivative modular development plan flexibly supported multi-site development. To effectively use a second development site, the project needed clear, simple division of responsibilities. The Model-View-Controller design pattern and constraints caused by the *syngo* application framework led us to organize the modules of the system into ten major subsystems, as was seen in Fig. 3.

Project Plan

The initial project plan contained five major milestones. The following table lists the work to be completed at each milestone for both the common assets and for the first two toolkits. (The VF Toolkit refers to the Ventricular Function imaging protocols.)

Table 1. Major Milestones

Milestone	Common Assets	VF Toolkit	Flow Toolkit
Mock-up	GUI Look-and-feel	Complete GUI mock-up	Complete GUI mock-up
First build (internal)	Partial framework	One complete scenario	GUI subset design
First trade-show demo	Complete Stand-alone Framework	Demo subset of functionality	GUI subset implementation
Third build (internal)	External interfaces, rework and testing	Full functionality and testing	Full functionality
First release	Rework and testing	Rework and testing	Rework and testing

This plan shows the concurrent design of the common assets and two toolkits, with the implementation work somewhat staggered.

Image Processing Performance

Image processing algorithms played a key role in the Viscard products. For example, in the ventricular flow toolkit there were three edge-detection algorithms: an algorithm that "improves" a hand-drawn contour by finding a nearby contour that fits the data better; a contour propagation algorithm that copies a contour from one image to another and uses it as an initial estimate upon which to improve; and, an interactive tool that continuously recomputes a contour that best follows the edge that the user is trying to mark.

There were many uncertain factors influencing image-processing performance, and no effective way to predict which of them would actually be problematic. New acquisition protocols might supply more images per series, or larger images, increasing the amount of data to be processed in a single operation. New quantification methods could increase the amount of work to be done on a series. New viewing techniques might require new image display algorithms. The expected performance of the new image processing libraries provided by *syngo* was also not yet known. Future hardware configurations

would be likely to have higher-resolution displays, requiring more interpolation. New image analysis techniques might help or hinder, either by making the same image quality available faster, or by making better image quality available, but more slowly. For example, image interpolation algorithms must be selected and tailored to find the right combination of speed and fidelity, while also maximizing the diagnostic value of the image. Ideally, the algorithms should all execute with interactive response time, but on larger sets of images this was expected to be impractical. New workstation processors would probably have higher performance and more memory, but it was not known whether processor speed and memory size would grow faster than computation and memory requirements. Finally, there was some discussion of using Viscard in soft-real-time contexts during image acquisition to help the technician navigate to the best coordinates for acquiring the images.

With so many uncertain influencing factors, the main design strategy was to make choices that would allow performance improvements to be introduced later with minimal impact as the critical factors became apparent. Already, the protocol framework strategy and the Model-View-Controller design pattern encapsulated many of the image processing algorithms. In order to allow some algorithms to operate asynchronously ("in background"), the project used asynchronous communication mechanisms – provided by *syngo* – between the model and view and between the model and controller. This made it easy to change the implementation of any particular Model method between synchronous and asynchronous operation. The *syngo* image processing library and image display API accepted parameters that trade off speed, resolution, and other image qualities. These parameter choices were encapsulated in the appropriate model and view classes. The properties of the user's display were similarly encapsulated in the framework view classes.

Experience

In this section we discuss and evaluate the product-family architecture and planning strategies for these applications.

Exploratory Protocol Development

The four main architectural strategies (protocol toolkit product family, protocol framework, Model-View-Controller design pattern, and reusable toolkit building blocks) effectively facilitated evolutionary protocol development. In addition, the decision to develop multiple toolkits within a single project and using a single team facilitated the discovery, design and development of common assets.

During initial development, GUI design iteration started early and continued until late in the initial development cycle without disrupting the development schedule. The prototype was done as a Powerpoint® mock-up, which, to our surprise and delight, the marketing manager began modifying in the field, in response to customer feedback. Prototyping two GUIs side-by side helped identify which tools would be common assets and which would be specific to a toolkit.

Near the end of the first implementation cycle, the marketing manager requested delivery of a new, stripped-down toolkit (called Cardiac Viewer) containing all the general-purpose tools in it, but no containing any protocol-specific tools. The architecture made it easy to design this toolkit, estimate its development cost, and insert it into the plan between delivery of the first and second protocol toolkits.

Investing in common assets has paid off even more in the third development cycle. The effort to develop the Flow toolkit was much less than expected because it reused so much design from the VF toolkit and code from the common assets. Two more toolkits are currently being developed. One was developed first as a WIP, and then split off as a separate project at a university. The other is an improvement to the original Ventricular Function (VF) toolkit. The WIP effort was higher than expected, apparently due to a substantial number of new staff. The VF improvement effort has not been estimated yet. These efforts both involve new UI viewing layouts, and adding and removing views. The viewing framework has been able to handle these variations without significant rework.

A particular way to display results, called a bull's eye plot, was first introduced in the Flow toolkit and implemented as toolkit-specific code. When the new WIP designer decided it should also use a bull's eye plot, the code was moved to the common view library and became a common asset.

The VF improvement adds a new image processing technique, *model deformation*. This is the first Viscard toolkit to use a true 3D representation of the heart or artery, instead of representing it as a stack of 2D slices. However, this difference is mostly confined to the Model portion of the toolkit; most of the views of the heart still look the same, although some of the controls will be different. The designer reports that the improvements fit comfortably within the existing toolkit framework.

Localizing External Dependencies

Shortly after the implementation phase began, GUI prototyping results indicated that a major *syngo* code asset would not meet the GUI requirements. This substantially changed the View and Controller framework designs, but not the Model, nor specific protocols.

After this major design change, the Viscard design successfully localized the impacts of instabilities in the image processing library, image attribute data formats, and startup and shutdown protocols. However, the plan failed to anticipate that dependencies on *syngo* and on the Microsoft Developer Studio would cause schedule disruptions.

The *syngo* image processing library caused instability both because of functionality and quality problems. The project discovered fairly late in the first development cycle that Viscard was the only early application planning to use *syngo*'s *cine* functionality. This was problematic in that the *syngo* project kept postponing it to a later release. In response to the delay, the Viscard project implemented a cine view, encapsulated in an image viewing class where it could be easily replaced later with *syngo*'s implementation.

Viscard was also the only early adopter of a certain "advanced" interface to the *syngo* image processing engine, which turned out to be less adequately tested and less stable than the more widely used, "basic" interface. Although the Model-View-Controller

design pattern limited the impact of these quality problems to the View framework subsystem, the impact was still quite substantial.

Viscard was developed concurrently with its MR modality host product, which was creating new, vendor-specific extensions to the DICOM MR image-attribute data formats. Even though the MR project made several late changes to these formats, the impact on Viscard was minimal. Later, when Viscard was extended to handle CT images, only a small amount of data handling code had to be changed to recognize the differences between CT and MR images.

Because Viscard was embedded within the *syngo* framework, it had to start up and shutdown on demand, under the control of the host product, which might be an MR or a CT image acquisition system or a multi-modality image review workstation. However, the project had to develop and begin testing Viscard before this control interface had even been designed. Therefore, separate subsystems encapsulated the toolkits, the sessions, and start up and shutdown. This allowed us to use a temporary, stand-alone version of the startup and shutdown subsystem until the control interface was implemented. Modifying that subsystem to use the new control interface, once it was available, was straightforward. Later, when the CT modality began embedding Viscard in its products, it used a different generation of the control interface than did the MR modality. Having encapsulated the interface, Viscard was able to support both generations without disturbing the rest of the code.

The Viscard project failed to anticipate that it would have to coordinate its *syngo* upgrades with the host product's upgrade planning. Besides the direct effect of interface changes, *syngo* upgrades also dictated when Viscard had to upgrade to new releases of development tools, because the *Syngo*, MR, and Viscard projects all had untracked dependencies on tool vendor libraries and vendor-specific language extensions.

Plan Flexibility

The project plan changed many times during the project without loss of control. Initial project planning created bottom-up estimates of the effort to develop each of the major subsystems, split out by common assets and toolkit-specific models, views, and controllers. This allowed evaluation of several alternative schedules. Monitoring effort-to-complete against expected staffing levels created an early warning system for possible missed milestones.

Two examples of the advantages of the flexible plan are, first, when the project manager and the customer agreed to postpone the delivery of the second toolkit (so that the first toolkit could be delivered on time with available headcount), they were able to adjust the plan easily because the effort estimates were directly mapped to pieces of the architecture. Second, when the marketing manager requested the new "Cardiac Viewer" toolkit, the product-family architecture made it easy not only to see which pieces would need to be added or changed, but to see the dependencies and the effort involved. As a result, the second toolkit delivered was one that was unanticipated at the time the architecture was designed, yet it had been readily planned, implemented, and delivered.

Skill Management

Technology overload was a major source of problems in the project, only partially mitigated by the architecture and project plan. As the architecture, high-level designs and detailed designs were being refined in UML class diagrams, project newcomers were increasingly able to learn or improve object-oriented development skills by reading and working from existing designs. For example, two months before completing the first internal build, the key developer of the Model (framework and first protocol) left the project. The image processing specialist who replaced him was delighted to find a set of class diagrams laying out the design. If the Model had not been separated from the View and the Controller, it would have been much harder to find and train a suitable replacement.

Outside consultants helped fill out the programming staff quickly, when needed. This strategy worked fairly well for consultants assigned to work on the Controller, which depends more on Microsoft Foundation Classes and less on *syngo*, but did not work well for programmers assigned to the View subsystem, which had substantial *syngo* technology dependencies. This is, in fact, a general problem with using proprietary programming technologies. The learning curve was also steeper earlier in the project than it was later, when there was more working code to study.

Because these subsystems were organized in part according to the programming skills and domain knowledge needed, it was effective and efficient to assign the development of different subsystems to different sites, occasionally moving a subsystem from one site to another over the course of the project.

The chief architect was also the View subsystem designer. This created a resource conflict which, combined with staffing shortages, left the architecture under-enforced during the initial development cycle. Later, when some architecture violations were eventually detected, they required more effort to repair than if they had been discovered earlier.

Implementation Processes

The product-family approach had some other interesting impacts on the development processes. Adding multi-site configuration management to all the other new technologies caused major disruptions. Besides switching over to new check-in and merge procedures, there were major difficulties getting the new tool to coexist with all the other new tools, as well as insufficient, on-site experience with them to quickly debug the problems. During certain periods of the project, many staff days were lost to damaged workspaces, corrupted run-time libraries and debugging multi-site synchronization processes. However, once these interactions were resolved, the code organization (which placed separate subsystems in separate directories and separate DLLs) allowed us to do twice-daily merges of the multi-site CM database without too many unpleasant surprises.

For the first few development cycles, testing everything together seemed rather efficient. By the fourth toolkit, however, retesting all toolkits at every delivery was becoming burdensome. When the second image acquisition modality, CT, was added to the product family, the CT development organization wanted Viscard to do a separate set of quality

assurance procedures to fit its process. Since the differences between the two modalities were minimal from Viscard's point of view, we arranged instead to have the MR organization deliver Viscard to CT, after passing its quality assurance tests, to avoid unnecessary duplication of testing efforts.

For Viscard, a single distribution kit was delivered to all customers and configured on-site. It was also eventually found to be cost-effective to have all installations run with exactly the same execution "footprint." The bulk of the footprint resources were taken up by the common assets, so creating different footprints for different combinations of products gained little execution efficiency but increased development and maintenance costs. Instead, the installation tools used run-time mechanisms to tailor the system, including license keys to make sure the customer only gets what she paid for. In the future, as more products are added, it may become cost-effective to create different execution footprints for different customers. If so, the architecture already has been designed to accommodate this change, and the component discipline used to separate different products will assure that the footprints contain only the components needed for the purchased products.

Image Processing Performance

In general, there were no surprising shortfalls in image processing performance. However, as expected, several of the image processing functions could not be computed fast enough to be considered "interactive". For example, *contour propagation*, which had to fit two contours to every image in the data set, in many cases took a large fraction of a minute to compute. Similarly, the cine view operates as a "continuous loop", which the user might want to leave running indefinitely while performing other steps of the analysis.

The initial response to the contour propagation issue was to spawn a background process to propagate contours, freeing the user to use other tools. However, many of the tools depended on the same data that the contour propagation algorithm was using, which could have caused either synchronization problems or confusing results. One prevention technique was to lock out the commands which could cause such problems, for example by "graying out" the corresponding controls and menu items. However, the application framework did not have a mechanism for disabling controls and re-enabling them later. Any such solution would have required either a major change to the framework or an ad hoc solution for each situation. Instead, the decision was to limit the user to just one action during the background computation. However, this is not a satisfactory long-term solution.

The underlying architectural mistake that was made was to describe each tool, conceptually, as a relatively autonomous package, but not to provide any framework support for propagating data between tools. Had the framework imposed a strict propagation discipline, such as publish-subscribe, and had the framework provided the means to disable some controls from time to time, then the user would not need to wait for computations like contour propagation to complete.

Discussion

We have described the process and results of the architectural design of a medical image-processing product family. This set of systems is a modest product family in that it consists of a set of sequential releases (four to date), each with significant additional functionality, but with each release delivering only one version of the software. Customization for individual customers is restricted to their choice of which protocol toolkit licenses to buy. We suggest that this sort of modest product family is common across many domains, and that our experience may be useful to builders of other such sequential, limited-variability systems in determining whether to follow a product-family design approach.

The benefits we anticipated when choosing to design the architecture as a product family were largely realized. These included architectural benefits such as extraction and modularization of previously unrecognized common behavior among the imaging modes, identification of features that were likely to change together, and reduction of unplanned dependencies. The realized benefits also included organizational factors, most notably the usefulness of the architectural design process as a mechanism to encourage communication among members of a distributed development team, and to support project planning and tracking.

More interesting were the unanticipated benefits—the lessons learned—that resulted from the design of the medical imaging system as a product family. Several techniques assisted in the development of an architecture that was fairly resilient to even unanticipated changes. The techniques that helped organize for change were the ones that demonstrated significant payback in this project. The most important of these product-family techniques were:

- **Investing in a thorough domain analysis.** The domain analysis helped produce a product-family architecture that was resilient to even many unanticipated changes (e.g., new protocols, the CT customer, the late delivery of the cine display library). Creating a playground for finding common functionality and migrating it into shared classes worked well, especially under the single-project model used here. By pulling out the common behavior into the protocol framework, the architectural dependencies among the features were minimized. As a consequence, the architecture was largely insensitive to the order in which the features were added. This also supported project responsiveness to customer feedback.
- **Decoupling product family architecture from order of implementation.** Making the architecture insensitive to the *order* in which features were added to the product was cost-effective and added significant flexibility. Minimizing architectural dependencies also allowed a single toolkit's schedule to slip without a major effect on the delivery of other toolkits. The decision to design the architecture so that different protocols were independent eased the subsequent development of different protocols by different parts of the organization. The decoupling of the product family architecture from the order of implementation also built in more project flexibility when organizational forces were not as

anticipated (e.g., staff reductions at one site, distributed development of protocols across different sites).

- **Using a target architecture to guide changes**. The development team did not fully implement the intended architecture in the first release, but used this architecture as a guide to subsequent improvements and reuse opportunities. DeBaud has described a similar pedagogical role for architecture as embodying "a standard that unifies a somewhat verified solution space while providing the concepts and vocabulary to communicate" [24]. In the product family described here, subsequent releases were closer to the planned architecture than initial releases. This is an interesting and encouraging inversion of the usual pattern of architectural drift, in which subsequent releases move away from an initial planned architecture. The use of the product-family architecture as a goal or "guiding star" for future development reflected the dedication to continued improvement by the development team.
- **Developing products sequentially to contain quality-assurance costs.** The need to certify each system encourages safety-critical product families to be built with only a few sequential releases and stringently limited variabilities. The overhead in time and cost of certifying a product, as well as the uncertainty of scheduling certification, makes the certification of multiple customized products difficult to accommodate. An alternative approach to certification of product families, used by Siemens in other systems, has been to certify the system generator and deliver it as part of each product. It is thus both the case that architectural choices affect the certification process ([25]) and that the requirements for certification influenced architectural choices, as here. More research is needed to better understand the reuse of certification results across product families.

Some of the lessons learned from our experience were negative, as well. Among the decisions and techniques that did not work well in this case study were:

- Basing the viewing architecture too heavily on an unstable external package.
- Failing to anticipate and encapsulate code dependencies on specific features of the Microsoft Developer Studio.
- Underestimating the impact of new technology overload.
- Placing too low a priority on architecture monitoring and enforcement
- Not initially providing sufficient support for background computations.

If these problems make any argument against using a product-family approach, it would be that the project was already too high-risk to introduce a new architectural approach. The principal alternative would have been to build an interim product in order to learn the new technologies and explore the architectural structure, then use a product-family approach on the next generation of the product. However, since the first release performed well in terms of cost, quality, schedule, and customer acceptance, the overall product-family approach was judged successful by the organization. .

Conclusions

The results described above for the medical-imaging product family indicate that a product-family approach to architectural design is, in fact, appropriate and cost-effective for modest product families with few releases and few variabilities. The benefits experienced are similar to those reported for architectural analysis and design of more ambitious product families. Additionally, the product-family approach tends to focus on issues that can affect future risk in safety-critical, modest product families: architectural independence from order of enhancements, architectural independence from organizational divisions, movement of the implemented architecture toward (rather than away from) the designed architecture, and the need for certification of each system in the product family.

Acknowledgments

We wish to thank Gareth Funka-Lea and Walt Borys for discussions regarding which strategies were successful in this case study, Dan Paulish and Rod Nord for their insightful comments, and anonymous reviewers for suggestions on improving the organization and focus of the paper.

References

1. M. Ardis, N. Daley, D. Hoffman, H. Siy and D. Weiss, "Software product lines: a case study", *Software Practice and Experience,* 30, 825-847, 2000.

2. J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*, Addison-Wesley, ACM Press, 2000

3. W. Lam, "A Case Study of requirements through product families," *Annals of Software Engineering*, March, 1998, pp. 253-277.

4. D. M. Weiss and C. T. R. Lai, *Software Product Line Engineering*, Addison-Wesley, Reading, MA, 1999.

5. P. Clements and L. Northrop, *Software Product Lines : Practices and Patterns*, Addison-Wesley, Reading, MA, 2001. A preliminary version is available at htttp://www.sei.cmu.edu/plp/framework.html

6. E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, Reading, MA, 1995.

7. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1998.

8. R. Lutz and G. Gannod, "Analysis of a Software Product Line Architecture: An Experience Report," *Journal of Systems and Software*, vol. 66: 3, 2003, pp. 253-67.

9. D. Paulish, Architecture-Centric Software Project Management. Addison-Wesley, Reading, MA, 2002.

10. D. E. Perry, "Generic Architecture Descriptions for Product Lines", *ARES II: Software Architectures for Product Families*, Gran Canaria, Spain, 1998.

11. R. W. Schwanke, "Layers, Decisions, Patterns, Styles, and Architectures", in *Proc. Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society Press, Los Alamitos, CA, 2001.

12. K. Schmid, "A Comprehensive Product Line Scoping Approach and Validation," Proc. 24th ICSE, ACM, Orland, FL, 2002, pp. 593-603.

13. P. Knauber, D. Muthig, K. Schmid, and T. Widen, "Applying Product Line Concepts
in Small and Meduium-Sized Companies," *IEEE Software*, 17:5, pp. 88-95.

14. S. Thiel and F. Peruzzi, "Starting a Product Line Approach for an Envisioned Market," in *Software Product Lines, Experience and Research Directions*, ed. P. Donohoe, Kluwer Academic Publishers, Boston, 2000, pp. 495-512.

15. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method," CMU/SEI-98-TR-008, Software Engineering Institute, Pittsburgh, PA, 1998.

16. F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi, "The Architecture Based Design Method," TR CMU/SE-2000-TR-001, Software Engineering Institute, Pittsburgh, PA, 1998.

17. Voget, Stefan and Martin Becker, "Establishing a Software Product Line in an Immature Domain," in SPLC2, ed. G. Chastek, LNCS 239, Springer-Verlag, Berlin Heidelberg, 2002, pp. 60-67.

18. Wijnstra J. "Supporting Diversity with Component Frameworks as Architectural Elements," 50-59. *Proceedings of the International Conference on Software Engineering*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000.

19. B. J. Pronk, "An Interface-Based Platform Approach," in *Software Product Lines, Experience and Research Directions*, ed. P. Donohoe, Kluwer Academic Publishers, Boston, 2000, pp. 331-352.

20. C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, Reading, MA, 2000.

21. Schwanke, Robert W., "Architectural Requirements Engineering: Theory vs. Practice," STRAW, 2003.

22. W. Pree, *Design Patterns for Object-Oriented Software Development,* Addison-Wesley Reading, MA, 1994.

23. J. van Gurp, and J. Bosch, "Design, implementation and evolution of object oriented frameworks: concepts and guidelines," *Software Practice and Experience*, 31, 277-300 (2001).

24. J-M. DeBaud, "Where Should We Invest for Reusability to Live Up to its Potential?", Symposium for Software Reusability, in *Software Engineering Notes*, 26, 103-106 (May, 2001).

25. P. Rodriguez Dapena, "Software safety certification: a multi-domain problem," *IEEE Software*, 1631-38 (1999).