The Application of Object-Oriented Design Techniques to the Evolution of the Architecture of a Large Legacy Software System

Jeff Mason and Emil S. Ochotta

Xilinx Inc.

The Application of Object-Oriented Design Techniques to the Evolution of the Architecture of a Large Legacy Software System

Jeff Mason (jeff.mason@xilinx.com) and Emil S. Ochotta (emil.ochotta@xilinx.com)
Xilinx Inc.
2100 Logic Drive
San Jose, CA 95124

ABSTRACT

Object Oriented Analysis and Design (OOAD) is increasingly popular as a set of techniques that can be used to initially analyze and design software. Unfortunately, OOAD is a relatively new concept and many large legacy systems predate it. This paper presents the approach one company followed in applying OOAD techniques to an existing 2.5 million line code base. We present an iterative process that provides an avenue for the software to evolve while balancing the needs of business and software engineering. Our case study reveals the many pitfalls that can derail a software re-engineering effort, but also shows promising initial results from continued perseverance in this effort.

1. Introduction

Object Oriented Analysis and Design (OOAD) techniques promise many benefits to software developers and software companies - software reuse and resilience to change through component libraries and patterns[1][2], lucid code structure that more clearly reflects the problem domain[3], and reduced risk by introducing a formal design process where often none existed previously[4] - to name only a few. To reap these rewards, most OOAD techniques assume software developers apply the techniques at the beginning of the software lifecycle, i.e., the beginning of the design process, and continue to use them as the software matures. Unfortunately, OOAD is a relatively new concept and many large legacy systems predate it. Moreover, because the pressures of commercial competition focus directly on adding features, fixing bugs, and releasing the product on-time, software developers often (misguidedly) skimp on the things that should be done for long term benefit in favor of the things that absolutely must be done to complete the product. Since not all developers are educated as to the benefits of OOAD it is often one of the things overlooked in the headlong rush to a software release. The long-term price of this behavior is a large body of difficult to maintain software, proving the well-known adage that the overriding cost of software is not its initial development but rather its maintenance. In this environment of legacy software and corporate pressure, reaping the benefits of OOAD seems a very elusive goal.

This paper describes the process one company undertook to re-architect their large legacy software system and begin reaping the benefits of OOAD techniques despite the constraints of continuing feature improvements and a strict release schedule. This six-step process is as follows:

1. *Analysis*: evaluating the current state of the legacy software;
2. *Goal Selection*: determining a set of goals to guide changes to the software and allow evaluation of the results;
3. *Key Concept Selection*: refining the goals into a set of key concepts based on business requirements, software engineering principles and object oriented analysis and design principles;
4. *Planning*: determining how best to apply the key concepts to the legacy software to allow it to evolve towards a system that satisfies those concepts;
5. *Implementation*: making it happen; and
6. *Measurement*: evaluating the effectiveness of the changes against the original goals.

There is a substantial body of research that focuses on the technical aspects of software evolution[5][6] and reengineering[7][8], and many of the technical ideas discussed in this paper have been described elsewhere in one form or another. The contribution of this paper is the description of the process we undertook and how we selected and satisfied key concepts that balanced the demands of business, the requirements of software engineering, and the OOAD principles we wanted to pursue. In the end, these key concepts included:

- ***Autonomy***: encapsulating and insulating functionally related software into subsystems to minimize interactions, to reduce compile times, and to support testing, allowing these subsystems to evolve independently and asynchronously;
- ***Sharing****:* solving problems in as few places and as few times as possible to maximize code reuse, minimize code size, and promote standardization;
- ***Comprehensibility***: promoting design, documentation and coding standards that - for the general client - make shared code and interfaces easier to understand, more convenient to use, and easier to maintain;

- ***Modularity***: allowing functional product components to be released to end users independently and asynchronously;
- ***Co-development:*** promoting the ability to explore, evaluate, and develop new features without affecting other on-going development;
- ***Innovation***: promoting runtime, memory, and quality of results performance through optimization and innovation;
- ***Testing:*** enabling efficient automated testing by creating a levelizable system[9] (i.e., a system where the testing and compile-time dependencies between software modules form a directed acyclic graph);
- ***Release***: supporting a release model with fixed release dates planned long in advance.

To implement these concepts, we developed a *system architecture vision* that outlined the changes to the software architecture that were designed to put these key concepts into practice. We then put forward an evolutionary plan to implement the vision. What quickly became apparent is that the inertia of the software was too large to allow all our changes to be implemented at once, while still releasing working software on an aggressive fixed schedule. Consequently, we realized that the six-step process described above must be applied iteratively, over an extended period of years.

Since the full implementation of this vision is an ongoing task whose costs and benefits may not be fully evaluated for many years, this paper describes the initial iteration through that six-step process. In this first iteration the implementation had to be scaled back to fit within a single release cycle of less than a year and focussed primarily on the key concepts of autonomy, sharing, testing, and comprehensibility. In these areas, we have seen some dramatic improvements, particularly where quantitative measurement is straightforward, such as compile-time coupling.

The remainder of this paper is organized as follows. In the next section we detail the first step in the six-step process we followed, outlining the state of the software system and the corporate situation that forms the backdrop for our work. In Section 3 and Section 4, we describe the next two steps in the process, the conceptual steps of setting the correct goals we are working toward and selecting key concepts that reflect those goals. In Section 5, we present the evolutionary plan we created to work towards realizing those key concepts in our software. In Section 6, we discuss the implementation of this evolutionary plan, and in Section 7, we evaluate this implementation against the key concepts and our initial goals. Finally, in Section 8, we present our conclusions.

2. Background and Analysis

In this section, we describe the first of the six steps in the process we followed to re-architect our legacy software system. We first present the environment in which our work was performed, including a brief description of Xilinx, the company where the work was performed, and the purpose of the software. We then discuss the state of disrepair we found when we first began to look at the software system itself and the costs associated with that disrepair. These costs were the initial motivation that drove our re-architecture.

2.1 Xilinx Inc.

This work was performed at Xilinx[10]. Xilinx was created as a hardware company, producing FPGAs, which are members of the family of integrated circuits (ICs) called programmable logic.

Understanding how an FPGA is used provides some useful insight into the complexity of the FPGA design software we discuss in this paper. An example FPGA application is emulating another IC or computer chip. In this application, the design to be emulated is loaded into the FPGA and the FPGA inserted into the system of which the chip being emulated is a part. This technique allows the design of the new chip and the system of which it is part to be tested and debugged before the new chip is actually built. Similar to a compiler, FPGA design software automatically translates the high-level description of the chip to be emulated into millions of programming bits that configure the FPGA to perform the emulation. Part of this translation task involves selecting a location from among the thousands available on the FPGA for each logical element. These locations must be selected to optimize chip performance or other user-specified constraints, creating an NP-complete[11] combinatorial optimization problem[12]. Moreover, in response to competition and customer demand, FPGAs are continually increasing in size and new hardware features are added to each new FPGA[13]. To keep pace with these newer, bigger FPGAs and still provide new software features, the FPGA design software is increasing in size and complexity at an even faster rate. Finally, because software provides the abstract model with which most FPGA customers interact, Xilinx has put increasing emphasis on software development in order to turn our software into a competitive advantage. The difficulty of the FPGA optimization problem, continually evolving FPGA hardware, and increasing customer reliance on fast reliable software conspire to make writing FPGA design software a challenging proposition.

2.2 The State of Xilinx Software

As one step toward improving its software, in early 1995 Xilinx acquired a small software start-up company based in Colorado. At that time, Xilinx’ FPGA design software consisted of nearly 1.5 million lines of C code developed and maintained by approximately 70 staff members. Xilinx had released 30 software revisions to over 10,000 software customers. In contrast, the 30 engineers of the close-knit start-up had written just over 700,000 lines of highly interconnected C++ and had released 6 software revisions to about 200 customers. The start-up's code was poorly documented, but a knowledgeable person was always at hand to deal with any issue or problem. Thus, change requests were informal conversations and system-wide changes could be implemented and compiled within a few hours.

After the purchase, the corporate goal was to merge the two software systems, keeping the strengths of both. This was easier said than done. The C++ code from the start-up was selected as the software base for the future merged product, and features that had been added to the original Xilinx product based on customer requests were to be added as needed. Software developers in Colorado were now faced with a much larger development environment and had to work with developers in California who understood the features to be added but did not understand the software base. Software developers in California were now faced with giving up their old software, learning a new and undocumented software base and working with developers in Colorado who did not understand the new features to be added. Neither group was used to working across multiple development sites, so "lack of communication" was one of the most common complaints by both groups about their peers on the other side of the mountains. Software was not getting built on time and fingers were being pointed in all directions. It was a difficult time for all involved.

Work toward the first merged release took substantially longer than anyone had dared to predict, and we missed several target release dates. Upper management began to apply greater pressure to the software team, justifying decisions to take "short-cuts" on the basis of short-term necessity. As is frequently the case, it is arguable whether these "short-cuts" reduced the time to first customer shipment, but they unquestionably came back to haunt us by adding to our maintenance burden over the next few releases.

After the frenzied days and nights of making our first few merged releases a reality, we took stock of our new software. The start-up's 700,000 lines of C++ had ballooned to approximately 2.5 million lines of C++ code in roughly 2200 source and header files. Our software shipped as 45 executables, 130 shared libraries (loaded at program start up), and 110 dynamically loaded libraries that customized the software for the different FPGAs in the Xilinx product line. Our single source software supported the Solaris, Windows, HP and RS6000 platforms. The source code was organized into approximately 400 subdirectories called packages, where each package produced either a library or an executable. After a brief inspection, we identified several major problem areas that we later categorized according to the key concepts to which they relate:

- **Comprehensibility**. Just as it was in the start-up, the code was mostly undocumented, but now it was much more complex and growing so rapidly that it was no longer possible to find any one person who understood most of it.
- **Autonomy (Encapsulation).** The interfaces between packages had evolved as necessary to meet tactical, local needs, without regard for strategic, system-level concerns. Consequently interfaces were extremely broad and ill defined. There was no clear division between the interface and the implementation of most classes. Much of the code was really just old C code transformed into C++ objects. One of the major indicators of a lack of encapsulation was direct access of class data by another class. Many of our classes had been designed with public get/set functions for each of the class data members. Consequently, changes that should have been internal to a package had repercussions throughout the system.
- **Autonomy (Insulation).** The compile-time dependencies (due to included files) had never been designed or analyzed. Often the vast majority of the compilation time for a module consisted of reading and processing included files. When first designing C++ classes, the tendency is to make the header file as convenient as possible for the implementation of that header. For example, the lowest level header file in the Xilinx software system directly or indirectly included almost 60 system header files, establishing a platform independent interface to the operating system. However, in such a large system, most of this functionality was not used by most of the clients that included it. This overhead is an unneeded burden to clients, who often compile complete definitions of many unused classes or classes that require only a forward reference. Engineers, recognizing this system-wide problem but unable to change it, were starting to make extremely large source files because the compilation times were faster than the aggregate compilation time of many smaller files.
- **Autonomy (Insulation).** Another problem was the rampant use of 'inline' functions. Inline functions are expanded at compile time rather than run time. This implies that a class that defines an inline function can not change the implementation of that inline function without forcing all of its clients to recompile.
- **Autonomy.** The turn around time to build and verify our software had become one to two weeks. Most of that time was spent in tracking down integration problems and then rebuilding everything. Because of the interdependence of the software, a compilation problem in one package may actually be caused by interface problems in any one of a large number of packages. Tracking down and solving these integration problems was made even more difficult because finding someone who understood the disparate parts of the system was no longer possible. Because of the difficulty of compiling several million lines of code on a single workstation, developers typically developed and tested against builds that were several

weeks out of date, exacerbating the integration problems for the next build.

- **Sharing and Autonomy.** There was no person or group whose responsibility it was to review or coordinate code changes. Each engineer or group was free to implement or use what they needed to get their specific job completed. Sometimes system-wide integration builds failed because large changes were made to shared code to support a new feature, but the changes were not tested for all clients. Other times, when small changes to a large package were required, engineers would copy the entire package into their package to avoid having to work with the other package's owner.

Problems like these were creating a software and corporate environment where developers no longer had the freedom or time to innovate. They had no freedom because every non-critical project was deemed high risk, since the complex package interdependencies could cause minor errors to have major repercussions throughout the system. They had no time because fixing each small problem required an inordinate amount of time to implement and verify.

After this analysis, it was clear that something needed to change. Fortunately for Xilinx, senior management understood the issues and that the long-term viability of the software product was at stake. With their support, several members of the company were chartered with re-architecting the software to fix these problems.

3. Goal Selection and The System Architecture Committee

The software management team recognized that Xilinx' software needed significant re-design at the architectural level, requiring co-operation from the entire software organization. They created the System Architecture Committee, a seven member team of engineers and managers that represented both development sites. The VP of software was a member of the committee, giving it the needed management clout. The authors were selected as members of this committee.

Initially, it was thought that members of the committee would spend roughly 10% of their time looking at system architecture issues, but as the weeks passed and the extent of the problem became more clear, the workload quickly grew beyond 10% of each member's time. To give the architectural work the attention it required, the authors became full time architects, and most members were required to put aside their other duties for short periods of time to complete work for the committee.

In the first few weeks of meetings, little was accomplished and frustrations grew. Several members proposed changes that they felt would improve the existing software architecture, but the group could not reach consensus. Eventually, it became clear that the goals of the various members of the committee were inconsistent, which led to disagreement over the changes that were required, which in turn led to stalemate and inaction. Consequently, the committee had to agree on its goals before it could take any steps to improve the software architecture. Choosing the goals was the seed for the six-step process that we eventually followed to bring our architectural changes to fruition.

The committee agreed upon six goals, several of which conflicted, making them impossible to satisfy all the goals simultaneously. Initially the group was disheartened that we could not select a set of goals that could be satisfied completely, but over time it became clear that this tension between the goals reflected the reality of business and of the software design process. In both environments, there are no right answers and compromise is essential to success. Moreover, the ability to strike the correct balance between competing goals is what distinguishes successful businesses and software organizations, and this made the design process challenging and exciting.

After much discussion and negotiation, the committee agreed upon the following six goals:

- *Provide superior end user productivity*. Make internal architectural improvements that eventually result in customer visible improvements in our software. Xilinx customers are the first priority.
- *Distribute productivity effectively across development groups*. Address “geography problems,” where developers in different groups do not communicate. The developers who were originally in the start-up felt they could not get their work done due to continually having to educate the other developers. The other developers felt they could not get their work done because they were not trusted to modify the existing core of the software. In practice, geography problems can happen even when the groups are physically adjacent, and the software architecture can have a significant effect on inter-group communication. These problems have a large negative impact on productivity and morale.
- *Improve the productivity of individual developers*. Create an environment where individual contributors can work more efficiently, without having to wait for other developers to complete their tasks.
- *Enable parallel development targeting multiple release dates*. Develop an environment that supports projects that require more time than a single release cycle. This goal is a direct result of the fixed release schedule required to support new FPGAs in a timely fashion.
- *Build in flexibility to handle a constantly changing market.* Anticipate the aspects of the software that will most likely change: new kinds of FPGAs, new software features, etc. Ensure that the software architecture is not brittle when these kinds of changes are required.

- *Enable accurate and efficient measurement of the quality of the system by designing for testability*. Plan from the outset to incorporate a testing infrastructure that supports measurement of software quality that is both fast and accurate.

These six goals formed the foundation for the rest of our software re-architecture work. They were driven primarily by business rather than OOAD or software engineering goals. When creating them, we also explicitly decided *not* to consider how we would accomplish these goals. They are merely what we wanted in an ideal world. Consequently, they form an ideal set of metrics with which we can evaluate the efficacy of our software architecture decisions.

4. The Key Concepts

Once the goals were in place, the next step was to determine how to achieve them. We soon realized that there was too large a semantic leap from the goals to actual architecture and code changes. What was needed was an intermediate step where we agreed on a set of principles from the worlds of OOAD and software engineering. These principles would reflect the above goals but more closely relate to the software and code architecture itself. This tighter relationship to the software would make it possible to create an implementation plan.

These principles are the eight key concepts introduced in Section 1 that tie our architecture work together. The first two (autonomy and sharing) are primarily OOAD techniques, and the last (release) is a business constraint. The others lie somewhere on the spectrum of OOAD techniques and plain old software engineering. As with the goals, these concepts are in tension: any plan will favor some concepts over the others. In this section we describe these key concepts and how they connect the six goals to changes that can be realized in a software architecture.

4.1 Autonomy

- **Autonomy**: encapsulating and insulating functionally related software into subsystems to minimize interactions, to reduce compile times, and to support testing, allowing these subsystems to evolve independently and asynchronously.

Autonomy follows directly from both of the productivity goals: *distribute productivity effectively across development groups* and *improve the productivity of individual developers*. Engineers are most efficient when they are free from dependencies and allowed to work alone or as members of a small, tightly-knit group.

Software dependencies can be exacerbated by poor software architecture and by failing to adhere to OOAD basics. For example, failing to encapsulate a data structure means that clients of a package use that data structure directly. When the data structure changes, the client must also change. This is an example of poor autonomy, since both the client and the supplier may be forced to wait for each other. The supplier may not be allowed to change the data structure until the client is ready, or the client may be unable to compile code that requires the new data structure until the supplier completes its implementation.

We recognize two facets of autonomy that are closely linked to OOAD principles: insulation and encapsulation. Insulation can be defined as the process of avoiding or removing unnecessary compile-time coupling[9]. In practice, insulation can be implemented by creating an opaque interface. For example, Lakos defines a fully insulated class as one that is not derived from another class, contains no inline functions or default arguments, and contains only a single pointer to an implementation class that is declared with a forward reference. The details of the implementation class are completely hidden from any clients that include the fully insulating class. The effect of full insulation is to create header files that are completely independent of each other, dramatically reducing the compile-time overhead of header file inclusion.

Another facet of autonomy is encapsulation, which should be familiar to practitioners of OOAD. Encapsulation can be defined as the concept of hiding implementation details behind a procedural interface[9]. Encapsulation and insulation are clearly related, but a fully insulated class need not be encapsulated. For example, a fully insulated class can still expose its implementation by providing public access functions to all its private data. However, in some respects encapsulation can be a less drastic technique than insulation because encapsulation allows the use of other features of C++, such as inheritance and inline functions. In this paper we refer to insulation when discussing the compile-time independence of modules from one another and refer to encapsulation when discussing the logical independence of a client class from the implementation decisions of its suppliers.

4.2 Sharing

- **Sharing**: solving problems in as few places and as few times as possible to maximize code reuse, minimize code size, and promote standardization.

Sharing falls into the general category of software reuse, a subject frequently discussed in the literature (see for example[15]). Reuse or sharing is also connected to the goal of developer productivity because in principle it allows a piece of code to be written once and reused in several places. In practice, sharing is difficult to achieve because the clients of the shared code must agree on what exactly the code does. If the code is too specialized, it is unlikely to be useful to more than one client. On the other hand, if the code is too general, it will be too slow or so simple that reusing it accomplishes little.

For the Xilinx software system, two kinds of sharing or reuse are of particular interest. Because Xilinx supports a number of different hardware devices that are fundamentally related, the Xilinx software is an excellent candidate for sharing via domain engineering[16]. In domain engineering, tasks that are needed throughout the domain are abstracted and written once. In this case, common tasks needed to support all devices can be abstracted and written as configurable or data driven algorithms. Fortunately, this characteristic had been recognized by the original designers of the core software created in the start-up and the software already made significant use of this kind of sharing (although the term domain engineering had yet to be coined).

The second kind of sharing was not as well supported in the Xilinx software, and that is the more conventional sharing of small generic algorithms. In sharing of this kind, tasks that are not domain specific, but may be general mathematical functions, data structures, or other algorithms are collected into a reusable library. To succeed at this sharing, this library has to be carefully designed explicitly so that it can be reused. The designers have to pay particular attention to making the functionality general, efficient, and well documented.

4.3 Comprehensibility

- **Comprehensibility**: promoting design, documentation and coding standards that - for the general client - make shared code and interfaces easier to understand, more convenient to use, and easier to maintain.

Comprehensibility as defined by this key concept is not intended to increase the amount of communication between developers, but to reduce the need for it. This key concept again relates back to the productivity goals. The idea is to create a system and an environment that inherently reduces the need for additional documents to describe the architecture of the system itself. One of the main benefits of such a system is the reduced need for maintenance that can occur when a change to an interface must be made both in code and in one or more separate documents.

4.4 Modularity

- **Modularity**: allowing functional product components to be released to end users independently and asynchronously.

Modularity is related primarily to the goal of *superior end user productivity*, but is an existing strength characteristic of the Xilinx software. As an example of this key concept, software support for a single Xilinx device could be shipped as part of the overall software system or as an individual software plug-in. This modularity made it possible to create and support new hardware products without shipping a complete new software system.

4.5 Co-development

- **Co-development**: promoting the ability to explore, evaluate, and develop new features without affecting other on-going development.

The key concept of co-development has two aspects that relate to what is being developed concurrently. Both relate to the goal of flexibility in a constantly changing market. In the case of support for new hardware devices, co-development means that new hardware can be supported with a minimal impact on software. This is essential in a competitive marketplace where the most successful company is the one that can innovate and respond to change the most quickly. Similarly, the other aspect of co-development is support for features and changes that are not driven by hardware, but must be developed somewhat independently from the main body of software because they extend beyond a single release cycle.

4.6 Innovation

- **Innovation**: promoting runtime, memory, and quality of result performance through optimization and innovation.

The innovation concept follows directly from the goal of providing superior end user productivity, which is fundamentally tied to software performance. Superior performance can be achieved using two methods that are in tension. The first method is optimization. This can be thought of as tuning existing software to improve its runtime, memory, or quality of result performance. Tuning software can sometimes compete with OOAD design principles such as encapsulation. For example, exploiting the underlying implementation of a data structure can sometimes result in significant improvements in performance, but at a clear cost in encapsulation.

The second method to improve performance is algorithmic change. For example, a developer may be able to squeeze a few percentage points of improvement out of a bubble sort algorithm by changing array operations to pointers and making function calls in-line. However, changing to a quick sort will yield significantly greater runtime improvements for large datasets because quicksort has better algorithmic complexity.

The two methods of improving performance are in tension because detailed optimizations that increase coupling of client algorithms to supplier algorithms also make it extremely difficult to innovate by changing either the client or the supplier algorithm. In most cases, algorithmic innovation yields greater improvements than optimization, so the focus of this key concept is on enabling innovation.

4.7 Testing

- **Testing**: enabling efficient automated testing by creating a levelizable system[9] (i.e., a system where the testing and compile-time dependencies between software modules form a directed acyclic graph).

The testing concept corresponds directly to the testability goal. In this case, the concept has a technical definition that can be concretely evaluated. By building a graph from the compile-time dependency structure, the system can be evaluated to see if it is levelizable. If there are any loops in the dependency graph, the system is not levelizable and is more difficult to test. This is because all modules involved in a loop must be tested together as a single unit. In the worst case, all modules will be involved in a loop and the entire system must be treated as a monolithic black box for testing. Since the difficulty of testing a module grows exponentially with the size of the module, creating a levelizable system is a desirable property. In a large system such as the Xilinx software system, it is easy to accidentally create a dependency that creates a loop in the compile-time dependency graph. The size of the system also makes such loops especially expensive in testing time.

4.8 Release

- **Release**: supporting a release model with fixed release dates planned long in advance.

The release concept is closely tied to the goal to *enable parallel development targeting multiple release dates*. An additional aspect of the release goal is to force the development to happen gradually in an evolutionary fashion. By requiring customer releases on a fixed schedule, the development is forced into an evolutionary path, which reduces schedule risk.

In summary, the creation of these goals and key concepts was a long and arduous process. However, because the key concepts provided techniques to realize the goals, subsequent work went significantly faster. Each new idea could be readily compared with the goals and concepts we had already agreed to implement, helping to keep the re-architecture process on track.

5. Planning

Armed with the newly created sets of goals, the key concepts, and a common mindset, the system architecture committee began to look at the software and come up with concrete plans for what should be changed. Here again, we followed a process that is clear in retrospect but at the time seemed full of bumps and blind alleys. We began by evaluating the current architecture against the goals and key concepts. We then chose the key concepts to be given first priority in the redesign effort. Based on the highest priority key concepts we proposed several different architectural solutions intended to address these concepts, then collected the best features of these proposals into a coherent document called the *system architecture vision*. With the vision as an endpoint, we created a plan to evolve from the starting point of our existing software architecture. Finally, we imposed the constraints of having to support new FPGAs, add new software

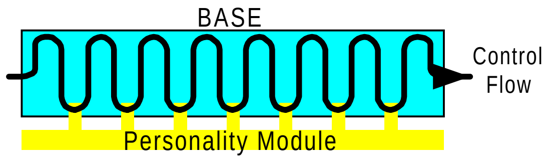

Figure 1. The Personality Module (PM) contained device-specific code that plugged in to the base code, but control flow was determined by the base.

features, fix bugs, and work with limited resources and a fixed release date. Considering these constraints, we then extracted a detailed short-term plan that would get us through the current release cycle. In this section we describe each of these planning phases in greater detail.

5.1 Prioritizing the Key Concepts

Before we could come up with an implementation plan of attack, we first needed to prioritize among the key concepts and decide which of them needed the most attention. To do this, we performed a careful evaluation of the existing software architecture against the goals and key concepts we had labored over for so long. This task allowed us to see which of the key concepts was least supported in the current software, and then to decide how we should focus our redesign efforts.

Based on this analysis, *autonomy* emerged as the most important of the key concepts to guide changes to the system. Secondary emphasis was given to *sharing*, *testing*, and *comprehensibility*. This ordering did not discount the importance of the other concepts - indeed the final solution would need to balance among all eight - but it recognized that the existing architecture already had certain strengths. The existing source code architecture was composed of two levels of hierarchy. The first level, called the Personality Module (PM) reflected the hardware device supported by that part of the software. Each PM contained packages, grouping the software within a PM by logical function. The remainder of the code, shared by all PMs, was called the “base”. This organization inherently supported sharing and re-use of base code by all other PMs. Moreover, each package from a PM created a Dynamically Loaded Library (DLL) that was loaded on demand, once the base code determined the device and required functions. As shown in Fig. 1, the DLL for the PM plugged into the base software, customizing it for a given hardware device. This meant that if the base required no changes an entire PM could be developed independently of the rest of the system (the co-development concept) and shipped to customers separately from the rest of the software (the modularity concept). Finally, much of the tight coupling in the system was done in the name of performance optimization. This tight-coupling was a two edged sword however, allowing significant performance gains via detailed optimization on one hand but on the other hand stifling the creation of new algorithms that promised leaps in performance. Here again we thought that increased autonomy was the key as it could increase encapsulation and make it easier to innovate algorithmically.

A secondary focus was the need for additional sharing. As already described in Section 4.2, the combination of base and personality module was an ideal situation for domain engineering, so there was significant sharing because the base code was reused for every device. However, there was no natural place in the system for algorithms and data structures that did not belong to any particular PM, yet did not define a new application for the base. Additional sharing of these generic algorithms was a secondary consideration for the architectural redesign.

Significant additional improvement was also desired in the area of testing. Within the existing architecture, anything not in a PM was added to the base, resulting in a very large base. Within the base, there were no rules about compile-time dependence and several packages were involved in compile-time loops. Also, we determined that significant gains in testability could be achieved by re-factoring the software according to function and designing from the outset a system that could be tested incrementally in sections.

Finally, we sought improvement in comprehensibility. Because the system had evolved into more than 400 packages, it was impossible to find a single person who had even a cursory understanding of the role of each package in the system. This made learning the system difficult for new developers, made tracking down integration problems difficult, and made it almost impossible to consider any large scale decisions about system structure.

5.2 The System Architecture Vision

Based on our priorities for the key concepts, we began to create a vision of where the software should be headed to better address autonomy, sharing, testing, and comprehensibility. This vision contained several elements and extended into the far future. Consequently, only two of these elements played a significant role in this first iteration through the six-step architecture re-design process. These elements were the re-organization of the source code into subsystems and layers and the creation of a special layer for generic algorithms. This section discusses these concepts in further detail.

Many long hours of discussion went into the creation of the vision document. After failing to write anything collectively, we delegated the task of an initial vision to one committee member. Having this draft allowed us to work through many problems and refine the concepts. After several iterations we completed an initial draft. We further refined the document based on feedback from a group of the top engineers in the software organization. Finally, the first version of the system architecture vision document was published and presented to all engineers.

However, the planning work did not stop there. The vision encompassed work that could take years to complete, but the next release was less than one year away. Moreover, with each release we had to support the latest hardware devices and offer improvements in features, runtime and software quality. After many hours of negotiation with marketing, sales, application support and senior management we reached a consensus on the resources that could be spent on re-architecting the software system. Matching available resources against the system architecture, we determined that we could make two major architectural changes: the addition of support for generic algorithms and the re-structuring of the software into layers and subsystems. Of these, the re-structuring of the software was a significantly larger investment. We describe these two changes in turn.

5.2.1 Layers and subsystems

Re-structuring the system into subsystems and layers called for a complete re-organization of the system from PMs and packages. It called for the creation of new units of functionality called subsystems, which would in turn be collected into layers.

On the surface, the software was still organized into a two-level hierarchy. However, the subsystems were envisioned as very different from the packages they replaced. These differences included the following:

- A Subsystem is typically larger than a package and can produce multiple libraries or executables. The structure within a package was flat, but a subsystem is truly hierarchical.
- Subsystems are logically related pieces of code that can have multiple people working on them. A single subsystem contains the base code and all the PM code for a single application or function.
- A subsystem provides a single directory that contains the *subsystem interface*, *i.e.*, any files exported by that subsystem, including header files, data files and libraries. Other subsystems can access only those files explicitly exported.
- Developers are encouraged to encapsulate and fully insulate their subsystem interface.
- New code in subsystems must follow a naming convention that limits pollution of the global namespace.
- Subsystems can have a compile-time dependency upon another subsystem only if that subsystem is in the same layer or a layer listed as a supplier layer. The graph of compile-time dependencies within a layer must not contain any loops.
- Part of the subsystem interface is a subsystem definition document that describes the subsystem and each exported header file. To avoid synchronization prob-

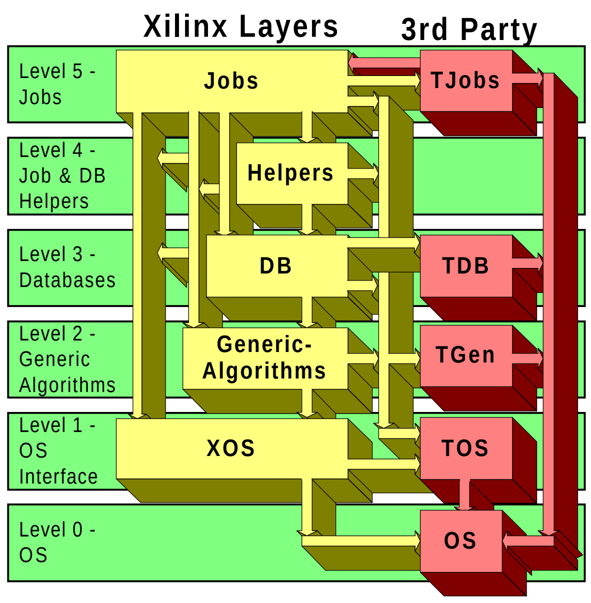

Figure 2. Include file relationships of Layers as put forward in the SAV document.

lems, the header file documentation is generated from the source code by ccdoc[14].

- The interface to a subsystem is controlled and changes require the approval of a committee. Synchronization of the changes within subsystems is handled differently than synchronization of changes in the subsystem interface. Each subsystem has a subsystem integrator, a new role that makes sure the different engineers working on the subsystem communicate. By having a single person responsible for each subsystem we expect to eliminate some, but not all, of the final build integration problems.

Of the changes proposed in the system architecture vision, these changes were perhaps the most directly connected with OOAD techniques.

Subsystems are grouped into layers. A layer is a set of subsystems with certain compile-time dependency or access rules. All subsystems within a layer have to abide by the access rules of that layer. As shown in Fig. 2, the entire Xilinx software system is composed of ten layers. Five layers contain code that originates within Xilinx, and the other five layers contain code that originates outside of Xilinx. In the figure, the arrows show access. For example, of Xilinx subsystems, only those in level 1 can directly include system header files. The principal purpose of access rules is to improve testability. However, the access rule that limits access to the system header files has the added benefit of making it easier to port to a new development platform.

5.2.2 Generic Algorithms

The second change to the software architecture is also apparent in Fig. 2. This is the addition of the generic algorithms layer. The generic algorithms layer encourages code sharing and reuse for numerical algorithms, data structures, and other generic algorithms. In addition to the code sharing between base and PM code (now contained within a subsystem) the intent was that the generic algorithms could be used for varied tasks throughout the software.

5.3 How the Architectural Vision Implements the Key Concepts

Recall that for this first iteration of improvements to our software architecture, we focussed on the key concepts of autonomy, sharing, testing, and comprehension. To work toward better support for these concepts, we chose to implement two major changes from our system architecture vision: a generic algorithms layer and the re-organization of the system into subsystems and layers. In this section we describe how the four key concepts on which our re-architecture effort focussed were realized through these two major architectural changes.

5.3.1 Autonomy

Since our engineering organization was split between two distinct sites, and a number of remote engineers were also involved, we felt that a rearrangement of our software along functional lines could help us provide better support for autonomous code development. Repackaging the software into subsystems created modules that are more self-contained and can be worked upon independently of other pieces.

Where modules are dependent on one another, greater stability is ensured by encapsulation, insulation, and committee approval for interface changes. Although some engineers felt these restrictions on the interface were overly harsh, we decided the best way to control the overall software structure was to control the subsystem interfaces. Ideally, we would also guarantee that the exported functionality behind the interface was also stable. For example, we could try to ensure that the return codes for a function do not change. However, in practice this is impractical and we rely on engineers to handle this level of detail.

Encouraging developers to use encapsulation and insulation techniques for their subsystem interface was a direct step towards improving those aspects of autonomy. As discussed in Section 4.1, complete insulation would forbid techniques such as inline functions, inheritance and default arguments[9]. We performed some simple tests and found that in certain cases changing a small set of inline functions to be called functions could cause a significant run time penalty. Since several of our applications can run for many hours, this performance penalty was not acceptable. Similarly, we have a reliance on inheritance that is used as the principal mechanism for writing code in dynamically loaded libraries. Because of these constraints, we could not require full insulation for all subsystem interfaces, but left it initially to engineer's discretion, subject to external review.

We further limited compile-time coupling by putting access rules on layers. The access rule with the greatest impact was that for Xilinx code, only the bottom layer could directly include system header files. This made designing the bottom layer more difficult because its exported files also could not include system header files, otherwise the system header files would be included indirectly by unknowing clients and insulation would be lost. We already had a set of system utilities that supported most of the system functions that might be needed. By making this rule we could speed up compilations and decouple the majority of our software from operating system quirks.

Encapsulation in our definition also includes keeping the global name space clean. In the absence of namespace support on every platform, every exported symbol must be unique across the system. This requires the creation of a system wide naming convention that can be uniformly applied. This also requires restrictions on the use of certain compiler features, like #define, that can potentially cause conflicts between subsystems. We looked for a tool that could be smart enough to help automate this clean-up process, but could not find one. In the end, we decided that much of what could be done in this area was impractical within the time and resource constraints of the initial re-architecture into subsystems and layers.

Consequently, although we did create a system wide naming convention that applied to all new code, we grandfathered existing code. This can lead to inconsistent header files that contain classes that follow different naming schemes. To compensate, we allowed the use of typedefs to make all classes within a subsystem consistent with the naming convention, but did not require clients of the existing interfaces to change. We determined that certain naming conventions had to be followed to avoid run time errors and required that these be followed, but other than that made few changes to the existing names of classes.

Despite these few exceptions left until later releases, we expect the introduction of subsystems and all they imply to lead to substantial improvements in autonomy.

5.3.2 Sharing

The introduction of subsystems and layers does little to effect code sharing. The base/PM relationship that implements sharing in the manner of domain engineering[16] is still supported by the new architecture. Now however, the base and its PM code are contained within a single subsystem.

The creation of the generic algorithms layer is aimed directly at improving the other kind of sharing, sharing of smaller more general purpose code. In this regard, the architecture changes are expected to lead to significantly more code sharing of these generic algorithms.

5.3.3 Testing

In the previous Xilinx software architecture, the separation of PM code from base code made it difficult to independently test the software. This is because the PM created a dynamically loaded library that was difficult to use in the absence of the base code. The concept of subsystems allows us to consolidate more of the code together and make the subsystem integrator responsible for maintaining and running tests. By having a single point of contact for a large set of code it was felt we had a better chance of getting a solid aliveness testing methodology in place.

The notion of levelizable software is also directly addressed by the architectural changes. Recall that if the compile-time dependency graph of a system contains no loops, it is said to be levelizable[9]. By organizing the subsystems into layers and strictly defining the access rules for layers, the system is likely to be levelizable. With the additional rule that dependencies between subsystems within a given layer cannot cause a dependency loop, we can guarantee that the entire system is levelizable.

By factoring the system into layers, we get the additional benefit that we can build and test the lowest layer first and then on up the dependency graph.

Consequently, the architectural changes lead to testing improvements in four areas: allowing the base and PM code to be tested together, making the subsystem integrator responsible for all subsystem testing, making it easy to guarantee a levelizable system, and allowing the system to be tested layer by layer.

5.3.4 Comprehensibility

A final contribution of repackaging the software into subsystems is the ability to provide a consistent and easy to use mechanism to learn about and understand our software. As our software grew we found it increasingly difficult to prepare developers to write new code. The learning curve was steep due to the lack of accurate internal documentation. Clearly more documentation was required, but large detailed documents are often imposing to new developers and poorly maintain by the author, making their value dubious at best. What was needed was a simpler approach.

To this end, each subsystem was required to provide a *subsystem definition document*. This document is created and maintained by the subsystem integrator in a standard HTML format. The document is not comprehensive because it does not deal with subsystem implementation. Instead, it briefly describes the purpose of the subsystem, the files and libraries it produced and its exported interface. Further documentation of the exported interface is a set of HTML documents that are generated directly from the exported interface header files using ccdoc[14]. In this manner a new or existing client of a subsystem can find an overview of the subsystem in the subsystem definition document, and detailed interface information in the header documentation.

We did not require that the subsystem implementation be described in an exported document, both because such a document would quickly become out of date and because it was “proprietary” knowledge not required by a subsystem’s clients. To engineers who typically looked at a function’s implementation before deciding whether it was what they wanted, this level of encapsulation and documentation was a revolutionary concept.

5.4 The Evolutionary Plan

After reaching consensus across the organization that the architecture would be reorganized into subsystems and layers, the final step was to schedule each of the packages to be converted into its corresponding subsystem. This task was made significantly more complex by the need to perform feature and device support work in the same time frame. In the past, Xilinx had tried to undertake major restructuring of its software, only to either fail or to wait many months before anything worked again. With the tight constraints of the release, neither of these risks was acceptable. Consequently we decided to convert the packages into subsystems in *waves*, changing only a fraction of the packages at a time. The exit criterion for each wave was defined to be working software that passed our internal engineering system test suite. The advantage of this process was that we would have working software after each wave. The disadvantage was that engineers had to create a different software environment for each wave, each with a different mix of old (packages) and new (subsystems). In order to mitigate risk, the plan introduced inefficiency by requiring almost everything in the system to change for every wave.

To prepare for each wave, the new subsystems were created several weeks in advance of the wave in which they were first used. This provided each subsystem with a trial period where it could be used locally but was not required for a wave to complete. To aid this mechanism we instituted a nightly build process where all the software released that day was built that night. These nightly builds gave us the chance to release a subsystem in one wave and then attempt to use it without affecting all existing clients in a full build.

In this section we discussed the changes to the software architecture and how those changes reflected the key concepts and goals for the re-architecture effort. We began by evaluating the current architecture against the goals and key concepts. We then chose the key concepts of autonomy, sharing, testing, and comprehensibility to be given first priority in the redesign effort. Based on the highest priority key concepts we proposed several different architectural solutions intended to address them and collected the best features of these proposals into a coherent document called the *system architecture vision*. We selected two of the ideas from the vision, deciding to create a generic algorithm layer and to re-factor the system into subsystems and layers. Finally, we created a plan to evolve from the starting point of our existing software architecture with limited risk. This plan called for the move from packages to subsystems to take place in several waves, ensuring that the system still functioned after each wave was complete.

In the following sections, we discuss the implementation of this plan and evaluate the effectiveness of our efforts to date.

6. Implementation

As of this writing, the initial transition to layers and subsystems is complete, and developers have been working with the new architecture for a few months. The wave plan succeeded initially, but after 3 waves, developers rebelled because the waves required a series of changes that needed to be re-visited for every wave. As a result, the final wave was more of a tidal wave that swept in all remaining changes. At that point, everyone understood the process well enough that we felt the risk involved in such a large change was justified by the time saved.

The major implementation hurdles to date have been more related to people and group dynamics than to technical issues. Initially, many engineers were somewhat confused as to what was actually happening, often because they were uninterested or too busy with other issues to take the time to really understand the process. Instead of riding the waves of change, unaware developers were hit by them, suddenly discovering that all their suppliers had new interfaces. Nevertheless, most of the work has been completed and we are not significantly behind schedule.

7. Results and Evaluation

As we said in Section 1, the re-architecture process is ongoing and will continue for many years as the software continues to change. However, we can begin to evaluate the initial two changes to the software architecture: creating a generic algorithms layer and re-factoring into subsystems and layers. These tasks are themselves incomplete, but preliminary results are encouraging. We detail our intermediate evaluation in terms of the key concepts these changes are intended to affect most.

7.1 Autonomy

At the time of this writing, it appears that there has been significant improvement in the ability of our engineers to work autonomously on their code. By separating the interface of a subsystem from its implementation and by placing controls on that interface, we have seen far fewer integration problems. Requiring engineers to

Table 1: Reduction in included files for a typical set of files

File (Level - see Fig. 2)	Num. Included Files Before	Num. Included Files After	Ratio: Before/ After
A (level 2)	44	6	7.3
B (level 3)	100	15	6.7
C (level 4)	217	97	2.2
D (level 5)	229	92	2.3
E (level 5)	327	223	1.5
F (level 5)	502	272	1.8
G (level 5)	266	110	2.4
H (level 5)	289	161	1.8
I (level 5)	321	156	2.1
Average			3.1

obtain approval for interface changes is cumbersome, but it makes engineers consider the impact of those changes.

In terms of insulation, results to date have been positive but depend greatly on the amount of time the engineer spent re-designing the interface to their subsystem. The first engineers to begin implementing their new fully insulated classes were greatly excited. After they spent weeks at a time working upon a single class and realizing that time was slipping away, the amount of insulation began to decrease dramatically. Engineers with less time often did almost nothing to redesign their interface. We have begun a detailed review of each interface to guide future work in this area.

In general, the engineers working on the lowest levels of the system (see Fig. 2) started working on their subsystems while other engineers were still working on the previous release. As a result, the most progress was made in the most heavily used portions of the system, which provides the most leverage to improve autonomy and reduce overall compilation times. This effect can be seen in Table 1, which shows the reduction in include file count for several files selected at random from various parts of the system. This is an important metric both because the number of included files is a rough measure of autonomy of a subsystem (more included files indicates less autonomy) and because including fewer files usually means faster compilations. These files were also selected because they are typical and because they have essentially the same functionality in the old system and the new. The levels in the table refer to the level numbering system shown in Fig. 2. Because of the additional work spent on insulating the lowest levels of the system, the most dramatic ratios of the numbers of included files can be seen for files in levels 2 and 3. For levels 4 and 5, more files are included because the code is at a higher level of abstraction. Moreover, most of the included functionality is from levels 2, 3, and 4, which have not been as well insulated. As a result, the ratio of the numbers of included files is not as dramatic.

Table 2: Reduction in compilation times for modules that are essentially unchanged

Module	Compile Time (s) Before	Compile Time (s) After	Ratio: Before/ After
A (level 2)	16	13	1.2
B (level 3)	44	25	1.8
C (level 5)	110	90	1.2
D (level 5)	513	335	1.5
E (level 5)	585	396	1.5
F (level 5)	75	50	1.5
Average			1.5

The reduction in the number of included files is also reflected in compilation time of the system. However, compilation times are difficult to compare both because the functionality of any significant subset of the system has increased and because computer hardware and networks are continually being upgraded. However, despite increases in functionality, compilation time for the complete system has been reduced from approximately 22 hours to approximately 5 hours. (The compilation happens in parallel, but the times quoted are the sum of the times from each machine.) Consequently, despite any hardware improvements, it is safe to say that compilation time has decreased.

Improvements in compilation time can also be seen in Table 2, which shows compilation times for several modules performed under controlled conditions: on a 200 MHz UltraSparc machine running Solaris 2.6. These modules were selected because they have remained relatively unchanged. It is meaningless to compare portions of the system where most of the improvements were made because the structure and function of the code is dramatically different. We can only extrapolate from the overall compilation times to estimate that the largest compilation improvements are in the re-written portions of the system. However, even when the module is essentially unchanged, because the subsystems on which the module depends has been insulated, the compilation time has decreased.

Although insulation can improve autonomy, it can also adversely affect the runtime of the application. In one case, several in-line functions were re-introduced into an insulated subsystem because of the overhead of the function call. In each case, the function was called so many times that the function call overhead consumed 1-

2% of the overall runtime of an application that ran several hours. Faced with this runtime overhead, the inline function was re-introduced. We plan to introduce alternative interfaces for clients with such special requirements.

As for encapsulation, the best we can say so far is that we have exposed our engineers to the idea of encapsulation. A few of the engineers did take this concept to heart and completely encapsulated their classes. These classes are now benefiting from this work in that they can have their class implementation changed without affecting their clients. Most engineers went into this project assuming that they were going to completely encapsulate every class. Once they started this process and saw the amount of time it took, they often retreated to insulation. This was acceptable because insulation provided immediately visible benefits that would encourage engineers to return to encapsulation as time permits.

The overall effect of the re-architecture effort on autonomy has been dramatic, particularly in the area of insulation. The use of encapsulation is more difficult to quantify, but we expect it will be more noticeable as the system continues to change.

7.2 Sharing

Improved sharing via the generic algorithm layer is also a long term investment. To date, several algorithms have been added to the layer and we have had at least one successful re-use. We expect greater utility over time, but unlike the optimistic predictions of early advocates of re-use, do not expect dramatic results from this effort.

7.3 Testing

Because we are not yet at that point in the process, we have not yet seen reductions in our testing burden or bug count. To provide a starting point for improved testing, there will be a special build, called a test build, used by developers to work on their internal tests. Each subsystem integrator will use this build to determine the amount of code coverage for their subsystem. Although we have tried to increase code coverage in the past, tight coupling with other developer's code was often used as an excuse for poor test coverage We believe that most engineers will be horrified by the lack of coverage and spend time increasing it. With this initial code coverage benchmark for the new software architecture, we can require increases in test coverage in future releases.

7.4 Comprehensibility

As with the other key concepts, comprehensibility is difficult to quantify and results have been mixed. In terms of documentation, we have created an on-line internal tools documentation area that contains the subsystem definition document for each subsystem. We have installed the ccdoc tool that creates documentation from C++ header files. Developers need only provide a specified type of comment in their interface header files and these will be added to the generated documentation. Even with all of this working, we hear anecdotally that not many people are using the browsing facilities. This is likely because most engineers have spent years

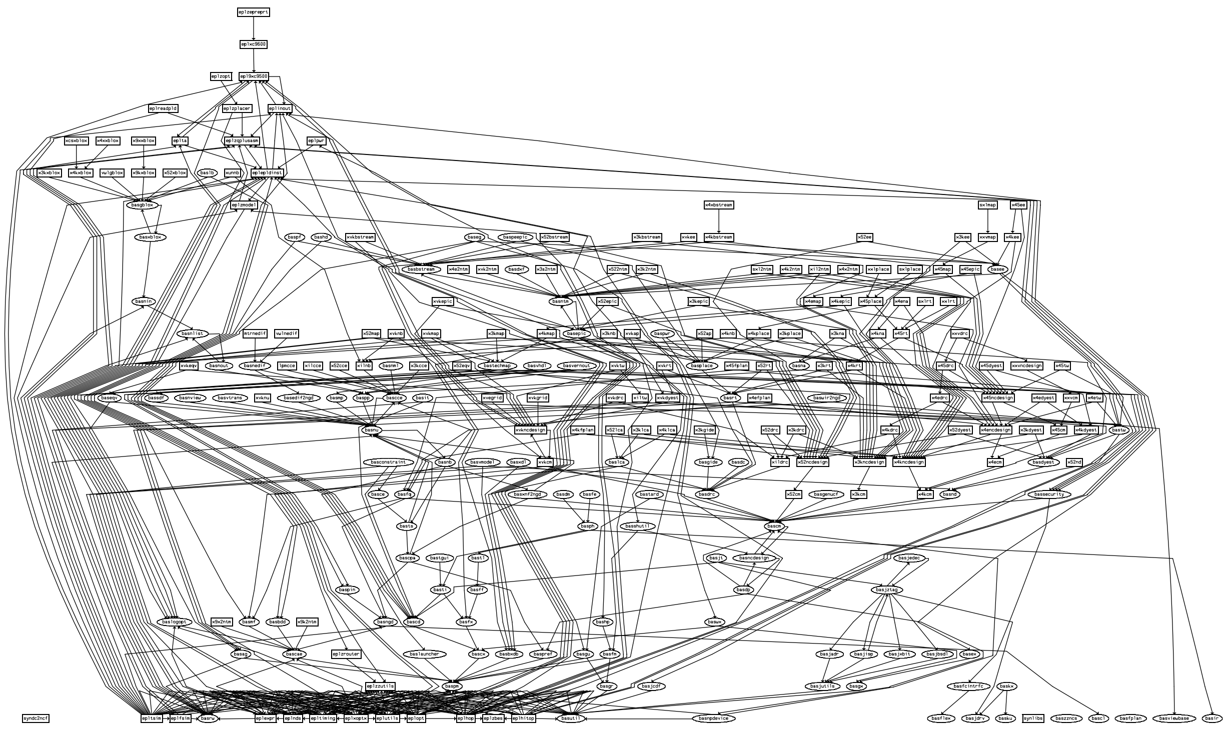

Figure 3. Topologically sorted graph of package include dependencies (before re-architecture).

reading header files directly and are still most comfortable doing so. Perhaps with the arrival of new engineers, this new browsing facility will become more useful. Similarly as we begin to move toward a heterogeneous mix of C++ and Java we might find these facilities more widely used.

One kind of comprehensibility where improvement is more readily apparent is the documentation of the overall architecture of the system. Drawings such as Fig. 2 provide a simple, high-level picture of the intended architecture. A complete drawing of all the subsystems and their include dependencies shown in Fig. 4 provide a more complete view of the system architecture. This graph is topologically sorted, which reveals that the compilation order of the system can be mapped back to the layers of Fig. 2. Note also that redundant edges (a direct edge to a subsystem included indirectly) have been removed. Fig. 4 also shows two remaining cyclic dependencies in the compilation order (upper right). These are in an isolated part of the system and will be eliminated soon.

Comparing Fig. 4 with Fig. 3, which was generated in the identical fashion from the previous architecture, it is quite apparent that the new architecture is easier to comprehend and work with at this level of abstraction.

8. Conclusions

In this paper, we introduced a six-step process by which Xilinx has made an initial iteration at re-architecting its software system. We have followed the progression through these steps and discussed the flow from analysis, to goals, to key concepts, to planning, to implementation and finally to evaluation. We have shown that we can modify a large legacy software system and create a software architecture that better balances among key concepts that reflect the demands of business, requirements of software engineering, and OOAD principles. Although this first iteration of our re-architecture is not yet complete, already we have seen significant gains in developer productivity.

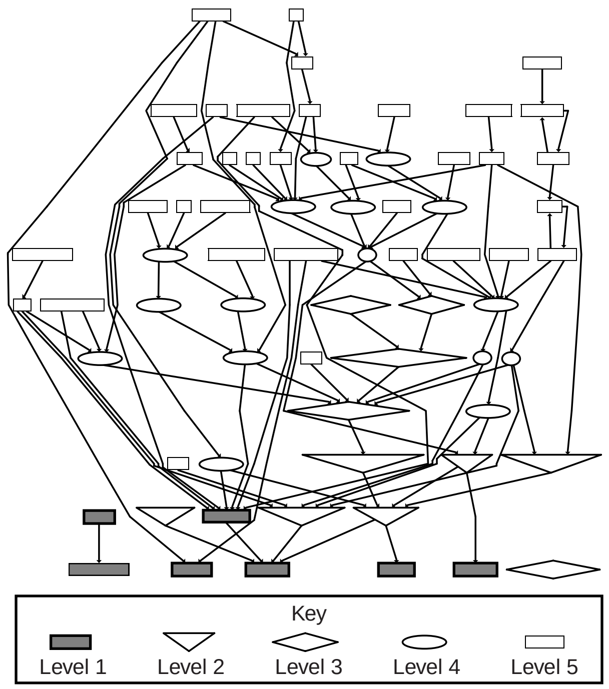

Figure 4. Topologically sorted graph of subsystem include dependencies (after re-architecture). Levels refer to Fig. 2.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] B. Foote and W. F. Opdyke, "Lifecycle and Refactoring Patterns that Support Evolution and Reuse," First Conference on Patterns Languages of Programs (PLoP '94). Monticello, Illinois, August 1994. *Pattern Languages of Program Design*, edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995

[3] G. Booch, *Object-Oriented Analysis and Design: with applications*. Benjamin/Cummings, 1994.

[4] S. McConnell, *Rapid development: taming wild software schedules*. Microsoft Press, 1996.

[5] http://www-cse.ucsd.edu/users/wgg/swevolution.html

[6] http://www.bell-labs.com/user/hpsiy/research/evolution.html

[7] http://www.comp.lancs.ac.uk/projects/RenaissanceWeb/

[8] http://www.sei.cmu.edu/reengineering/

[9] J. Lakos, "Large-Scale C++ Software Design," Addison-Wesley Professional Computing Series

[10] http://www.xilinx.com/company

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA, 1979.

[12] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," Proceedings of the 29th ACM/IEEE conference on Design automation conference, 1992, Page 536.

[13] E. S. Ochotta, *et al*, "A Novel Predictable Segmented FPGA Routing Architecture," in FPGA '98, Proceedings of 1998 ACM/SIGDA intl. symp. on FPGAs, pp 3-11.

[14] http://www.joelinoff.com/ccdoc/index.html

[15] C.W. Krueger, "Software Reuse." ACM Computing Survey, vol. 24, no. 2, pp. 131-182, 1992.

[16] E. Mettala and M.H. Graham, "The Domain-Specific Software Architecture Program," CMU/SEI-92-SR-9, 1992.