# Modular, layered architecture: the necessary foundation for effective mass customisation in software

Marc H. Meyer*

Northeastern University
212 Hayden Hall, 360 Huntington Avenue
Boston, MA 02115, USA
E-mail: MA.MEYER@neu.edu
*Corresponding author

Peter H. Webb

The MathWorks, Inc.
3 Apple Hill Drive, Natick, MA 01760-098, USA
E-mail: pwebb@mathworks.com

**Abstract:** This paper posits that developers cannot sustain extensive customisation of software without clear software product line architecture, modular subsystem platforms, and disciplined interfaces among these platforms. To explore this proposition, we initially define the basic principles of desirable software architecture and then apply these definitions to the development of software and the business strategy for creating software product lines. We then illustrate how these concepts not only explain the success of leading software companies but also have motivated fundamental architectural redesigns of their product architectures.

**Keywords:** modular architecture; software platforms; software development.

**Reference** to this paper should be made as follows: Meyer, M.H. and Webb, P.H. (2005) 'Modular, layered architecture: the necessary foundation for effective mass customisation in software', *Int. J. Mass Customisation*, Vol. 1, No. 1, pp.14–36.

**Biographical notes:** Marc H. Meyer is Director of the High Technology MBA Programs at Northeastern University in Boston, MA and Professor of Management at Northeastern University in Boston, MA. He was the 2002 recipient of the Holland Award from the Industrial Research Institute. Author of *The Power of Product Platforms* (The Free Press, 1997), Dr. Meyer is writing a new book on leveraging technologies to new market applications (with the Oxford University Press). He holds an AB from Harvard, and an MSc and PhD from MIT. In addition to his scholarly pursuits, Dr. Meyer is a co-founder of several software companies and continues to work with large corporations on developing next generation systems and software.

Peter H. Webb is Principal Technical Specialist at The MathWorks (Natick, MA) where he focuses on software architecture, virtual execution environments, and the use of cryptography in language processing systems. He is currently researching frameworks for multi-source tool integration in software development environments and the impact of service oriented

architectures on rapid prototyping systems. He holds a BS in Computer Science and English from Tufts University, a MSc in Computer Science from the University of Illinois, and an MBA from Northeastern University.

## 1 Introduction

Software is a profound example of mass customisation wherein users (or programmers supporting users) tailor standard products for specific functions and to meet specific needs. In fact, software represents an extreme example of mass customisation in commercial products. However, the 'catch', is that software also requires a high degree of internal architecture discipline in order for that customisation not to result in unmanageable chaos.

How many readers have not built equations in Microsoft Excel or their own graphic templates in Microsoft Visio? The small business owner sets up his or her own business categories and reports in Intuit's Quicken or Quickbooks, and larger corporations expend substantial resources to customise software from SAP, Oracle, and others. Organisations are constantly building application-specific data structures within general-purpose database management systems, whether to track customers and manage better workflows. Even consumer-focused software, such as email or instant messaging, is increasingly customisable by individual users for visual appearance, greetings, and actions.

As these examples show, software is clearly among the most pervasive forms of mass customisation reaching into our lives. Such customisation is different by degree than the more standardised forms of variety found in automobiles, consumer electronics, or interior furnishings (Simpson *et al.*, 2003).

In fact, it is reasonable to conclude that the market success of a software firm depends on its ability to support customisation of its products. The reasoning behind a strategy of personalisation to individual and corporate needs for software products can be stated as:

- Software (large, complex systems in particular) is expensive to build even when its architecture is well done.
- No single set of software options is going to satisfy all a company's potential customers.
- Making multiple separate large software systems to satisfy an ever larger and more diverse customer base becomes increasingly, if not prohibitively, expensive.
- Software designed for mass customisation (and this is why good architecture is so important because it *enables* mass customisation) offers a much more cost effective alternative.

Readily customisable software can serve many specific market applications much more cost effectively. It can remain viable over a longer period by the flexibility provided to uses to modify certain aspects of the software to match their own evolving needs. Customisable software also provides the software development organisation with a powerful form of risk management: it can carefully observe how users develop new solutions based on its software, and wait patiently for the best of these additions to appear as worthy improvements for its next major version. All of these are important elements of competitive advantage in hypercompetitive software markets (Von Hippel, 1988).

At the same time, mass customisation can be a double-edged sword. Like any other technical project, implementing high levels of flexibility for users to modify their software can be done well or poorly (Pine, 1992). The irony is that the user expectation for customisation of standard software can quickly lead the software development entity down the path of internal chaos within its own products. Customisation will create high levels of complexity and cost, *unless* the firm has taken the time to create and periodically renew a well-defined, layered architecture for its software products (Mann, 2002).

Customisation to a standard piece of software must be performed within the context of an underlying infrastructure. That infrastructure, we believe, must be highly structured, its components tightly defined, and the interaction between these components equally well-structured and tightly defined. Otherwise, user customisation of large and complex software applications carries an inherent risk: modification of the software might destroy it. To minimise the risk of that corruption, the 'where and how' of user modifications to software for their own purposes must be done at levels within the software that do not reach down into and destroy the integrity of the base system.

This is our premise: *delivering the customisation in software so greatly desired by users, across all applications and industries, cannot be sustained over the long term without clear software product line architecture, modular subsystem platforms, and disciplined interfaces between these platforms.* To understand mass customisation for software, we must first delve deeply into the meaning and use of architecture within software. To do this, we will proceed in four steps of logic:

1 Define the basic principles of desirable software architecture.

2 Leverage that definition for software architecture in general to software product lines specifically, including the meaning and application of *platforms* within product line architecture.

3 Leverage both these general and specific articulations of concepts and principles to a *business strategy* for a software company, where over time a firm leverages its architecture and platforms to new market applications, each one of which requires not only new standard modules, but also personalisation and customisation by users within each new segment.

4 Illustrate how these concepts for software architecture, platforms, and strategy have helped a category-leading software developer – known throughout industry for its mathematical modelling and simulation software – to fundamentally restructure its software to accommodate both strategic growth and more effective mass customisation.

## 2 Basic architectural principles for software: layers with focused functionality and robust subsystem interfaces

Software is an engineered construct, much like an automobile or a building. As such, a software programme consists of many parts or *components*. Like an automobile or a building, software can be said to have an *architecture*: one or more organising principles that control how each component relates to and communicates with each other component (Jazayeri *et al*., 2000). Architecture structures an unorganised space to serve

one or more functions, just as an engine provides power, walls provide privacy and shelter, or doorways orchestrate the flow of motion. The structure provided by an architecture for a product line is inherently hierarchical. Power-trains comprise engines, transmissions, and exhaust systems; buildings consist of floors and floors are divided into rooms.

Similarly, a software application consists of *layers* of hierarchy, with each layer containing one or more components. The boundaries, or *interfaces*, between layers and components, like the walls and doorways in a building, control the motion of information through the application. Of course, there are almost an infinite number and variety of software products, but at the highest level of definition, there are software development tools, software applications developed with these tools, and the systems software, often called operating systems, upon which both tools and applications run. Each one of these software products – operating systems, software development tools, and software applications – has its own layers of technology. Interfaces connect the modules within each layer, and one layer to the layers above and below. A software product – be it an operating system, tool, or application – in which these layers and component boundaries are well-defined is said to be *modular* (Dilip *et al*., 1995; Sharman and Ali, 2004).

Using these definitions, we can outline the principles of good software design. A well-designed software product must:

- Possess clear and explicit organising principles, expressed as the software product's *architecture*. Since architecture is typically developed not just for one product, but also for a stream of related products, the architecture is best referred to as a product line architecture (Meyer and Lehnerd, 2004). To support customisation, this architecture must describe the mechanisms by which the software, and even the architecture itself, are allowed to evolve.

- Be hierarchically organised into *layers*. While it is perhaps easiest to think of these layers as layers of specific technology, we find it best to view them in terms of function (such as a real-time data acquisition layer, a database layer, logic or algorithm layer, or the graphical user interface). The reason is that multiple technologies may exist within or they can be used to implement the functionality of a particular layer. For example, FibreChannel, InfiniBand, and iSCSI are all communications technologies used for network connection modules in modern storage array systems.

- That each layer consists of a *modular* set of *components*, each with its own single function or purpose, and each offering a single robust *interface* for other modules to access that functionality. The FibreChannel, InfiniBand, and iSCSI communication protocols mentioned above are implemented as separate components within the communications subsystem of the storage device. Alternatively, consider a basic report-writing layer, which consists of modules for querying information from a database, for accessing a report template, for pumping the information through that template, and then formatting it for the user's particular printing device. Interestingly, the output of one module in this layer of functionality becomes the input for another module, and so on. This connection works best if there is only one clear way to pass data into any given module.

Developers can adjust software to changing user and computing environment requirements when the underlying architecture adheres to these guidelines. Since the function and interface of each component is well understood, one component can be replaced without affecting the functionality of other components; and as the architecture explicitly provides for extension and self-evolution, developers can easily add new functionality (Clements and Northrop, 1996; Bass *et al*., 1998).

Good architecture in software provides modularity across and among all the various components within the software. That means that the components can be combined readily in different ways to meet different purposes without adding substantial additional code to make these components work together. With this foundation of flexible components, developers can then add additional modules that provide the functionality needed for a specific application. Good architecture makes software reuse possible. Further, if the foundation of core subsystems is particularly well designed, software developers outside the company – be it other software companies or programmers in end-user companies – can also quickly write their own application modules that plug immediately into that foundation and use its functionality. This is how the most successful software companies have developed extensive collections of specific software applications made by third parties.

'Network externalities' can play an important role in how users perceive a software product or system (Park, 2001). The most common form of network externalities – on the positive side – is when the value of a product for an individual user rises in proportion to the number of other people who use it. The telephone is a classic example. One telephone has little utility, but a worldwide network of telephone users makes each individual telephone a very powerful tool. On the negative side, of course are viruses such as Trojan horses that invade a user's computer to infect others, and from these others, yet others.

> "*Good architecture as we have described it, leverages the potential of any given piece of software to have a synergistic effect with other software. Poor architecture builds walls between different pieces of software, making integration difficult and costly*."

Just as a car manufacturer offers a line of passenger cars or SUVs, a software company can also offer a family of software products, be they applications, tools, or systems software. The goal is to have these individual products share a set of common subsystems whose function and interfaces to other subsystems are defined by the architecture of the product line. This no different than Honda leveraging the same two litre, VTEC engine across many of its passenger cars and sport utility vehicles. Software product families, when well designed, can have dozens of such engines.

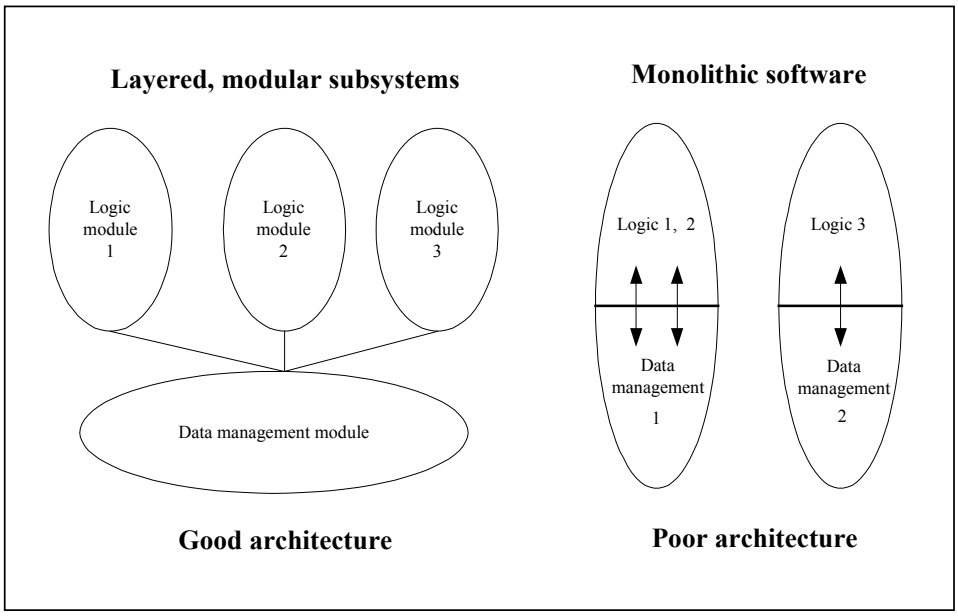
This leads to the two basic principles in designing product line architecture for software: subsystem focus and the use of single-function interfaces.

## *2.1 Subsystem focus*

Subsystem focus requires that each subsystem have a single, concentrated purpose. A focused subsystem is very different and much more useful over the long term than a nonfocused subsystem, because focus helps manage complexity. Without subsystem focus, changes in one part of the system tend to ripple through and impede functionality in other parts of the system. The goal of modular architecture is that changes at one level of the application do not destroy the integrity of modules at other levels.

A classic and still present example is application software that contains both logic – such as for business accounting or statistical processing – and data. On one hand, a developer might correctly choose to have a series of modules each containing a different set of accounting or statistical logic, which then communicate to another module focused on organising, storing, and accessing the needed data for the calculations. This reflects good architecture. Less desirable is a design where different buckets of logic are combined into one module, and even worse, where data management functions are also co-mingled with the logic in that module. This can be viewed as a smokestack architecture where each application is monolithic and shares little, if any, code with other applications in the product line.

**Figure 1** Rights and wrongs in software architecture



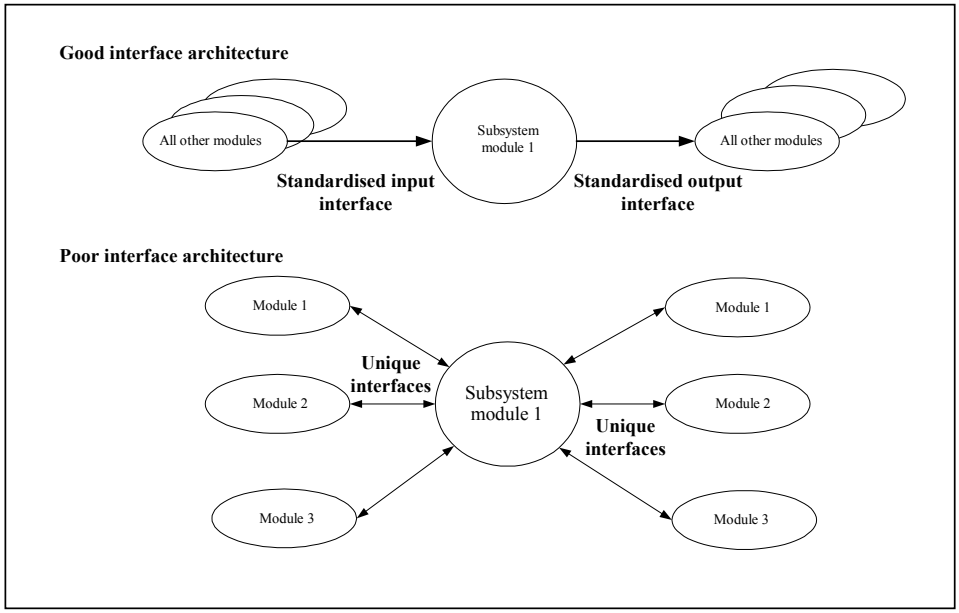

A smokestack type architecture, lacking clearly focused subsystems, makes the implementation of new technologies (such as a new data management method) labour intensive and error prone. It requires changes to many modules, and the developer must work doubly hard to insure that these changes do not impede the functionality of other code containing algorithms and logic. This can produce tremendous problems downstream for applications integration and maintenance. It is easy to fall into this trap if a developer tackles one new application at a time, fails to consider architecture at any point along the way, never stepping back to redesign the foundations across the product line to create common subsystems.

## *2.2 Subsystem interface design*

The second basic principle for achieving good architecture in software deals with interface design (Sundgren 1995; Meyer and Seliger, 1998). Each major subsystem within the product line architecture should have a single interface programme for data

coming into the subsystem, and for data exiting the subsystem, as shown in Figure 2. Both occur when the subsystem is 'called' or referenced by other programmes to utilise the subsystem's functionality. This has been called the module interconnection architecture (Soni *et al.*, 1995).

**Figure 2** Rights and wrongs in software interface architecture




This is in striking contrast to the way most systems tend to evolve over time, where developers build many input and output interfaces into each major subsystem. When the functionality of the subsystem itself changes, developers must then work through all the various interfaces to make sure that those improvements are used by other parts of the system. Figure 2 shows this evolving chaos at the bottom of the figure (Steward, 1981; Meyer, 1992).

The idea of using structured interfaces to link multiple subsystems is so fundamental to good architecture that it deserves a simple, yet powerful example for the reader. Historically, the idea took form in software during the 1970s. As IBM's mainframe software paradigm of hard-to-integrate, proprietary, batch computing spread through industry, an alternative model of easy-to-integrate, nonproprietary, interactive computing was taking hold in universities and research organisations. These users needed to develop a wide range of applications that could run on almost any computer at very little cost, and be readily integrated with other applications. Today, this is called 'open systems' computing.

Unix (including variants as Sun's Solaris and now open source Linux) met the needs of these users with a layered, modular architecture. Unix-based programmes can share data seamlessly both on single computers and across networks of computers. To allow this to happen, Unix utilises a specific modular architecture that works at two primary levels, data and programming. This is carried forward in Linux.

At the data level, standardised ASCII text is the universal representation for shared data among all programmes. A programme can expect to receive a stream of ASCII characters as input, and in turn, is itself expected to produce a stream of ASCII characters as output – at least until a final report is transformed into a binary format by a filter on its way to a printer. The data standard is flat ASCII files – simple streams of characters. The files do not have structure like the fixed-length records of the traditional transactions processing world. Instead, the ASCII files use a simple carriage return or new line character to mark the end of the record, and some other character, such as a tab or comma, to separate individual fields in the record. Data has a variable length and no space on a disk needs to be reserved ahead of time. All Unix/Linux programmes can expect to find this data structure, free of specific field length or record size predefinitions.

The second level of the universal interface is at the programming level. The output of any one programme can be 'piped' to any other, and the output of that second programme then piped to yet a third, and so on, until the user achieves his or her final objective. This approach finds its best expression in the higher-level command interpreters or 'shells' offered with Unix/Linux over the years. These command interpreters not only allow users to type commands to look at, change, or print files, but they also comprise full programming languages (Manis and Meyer, 1986).

Using these conventions for data and communication between processes, Unix/Linux facilitates the most personalised form of mass customisation at incremental cost. For example, if one were to type the date command in the Unix or Linux shell, the result would be:

```
$ date

Wednesday August 12 20:18:32 EDT 2004
```

Or, the result could be send to both a printer and a file with the 'tee' module, and using pipes:

```
$  date | tee mydatefile | lp
```

This command chains together three modules, *date*, *tee* and *lp*, to display the date on screen, write it into a data file and print it. The modules connect together very easily because they all use a common data representation (ASCII text), as well as common input and output channels (called *standard input* and *standard output*). *Date* writes ASCII text to standard output, *tee* reads it from standard input and copies it to standard output and *lp* reads it from standard input. The pipe command, |, creates interprocess connections among the three modules. This simple idea is infinitely scalable – filter chains like this can be of any length, and perform any type of computation – and adapts very well to a networked environment where server names could be used as prefixes for any file or device.

The power of the universal interface standard – at both the data and programming levels – becomes ever more apparent when the user works on increasingly difficult tasks. As a second example, consider two large customer data files, the first sitting on one computer with several months' worth of customer transactions, and the second, on another computer containing customer contact information. The Purchases database sits on one computer in the building; and the Customers database resides on another.

*Purchases Table:* Last Name, First Name, Middle Initial, Purchase Amount

*Customer Table:* Last Name, First Name, Middle Initial, Address, E-mail Address

A common request might be to produce a collections report. The logic of completing this task is straightforward. We sort these two files on customer name so that they can be combined without special indexing, and then sum up the amount owed by customer, and then sort it by purchase amount in descending order. Then we might want to save the result to a file so that we can send out letters to people, and e-mail the result to the manager of our collections department.

Implementing the simple logic of the solution in poorly architected software environments is unfortunately not so simple. We would hire a programmer who would first create several new interim databases and reserve space for them on the computer disk. Then, our programmer would write a page or two of C code to access data from the customer and purchases databases on the two different computers, store the data in the intermediate files, combine these files to produce the report, and then write systems level code to send reports through email to the collections department. Creating the linkages among databases across different machines and to an email system tends to be difficult in proprietary (*e.g.*, non-open systems) software environments. Overall, this might be a solid week or two of programming effort.

In Unix/Linux, the programmer can take a different track, seeking to leverage the simple, robust architecture of that environment with a few connected commands. After sorting the two files on customer names, he or she might type:

```
$ join /computer1/customers /computer1/purchases |
sort –r –4 | tee resultfile | mail accountsreceivable@ourcompany[1]
```

These commands could either be typed, or saved in a little command file that could be executed at a prescheduled time, or at the user's convenience.

Examples such as this show that developers can expand the functionality of software based on modular architecture, focused subsystems, and standard interfaces *at very little cost in programming effort or machine resources.*

## 3 Leveraging basic software principles to create software product lines

With these principles of software architecture in hand – a layered architecture comprising subsystems with focused functionality and disciplined, robust interfaces among them – a development organisation can then define an operational platform strategy for its software products.

Software, just like any other product category, is rife with terms that have different meanings for different people. The layering concept leads to the idea that subsystems in software products are programmed modules that have a clearly defined set of functionality, *i.e.*, a user interface, a certain set of logic, a data management function, printing, or communications (Sharman and Ali, 2004). It is also evident that software interfaces are the predefined connections between subsystems and other subsystems. In addition to subsystems and interfaces, other terms suffer from inconsistent use and often cause trouble between software development teams and management.

Common definitions for basic conceptual elements help empower a development group to achieve better designs. We define the essential strategic terms for software products as:

### *Product line architecture*

The Software Engineering Institute at Carnegie Mellon has defined software architecture as specifying the structural properties of a system in terms of components, interrelationships, and principles and guidelines about their use and evolution over time (Clements and Northrop, 1996). Missing in this definition is the purpose of the architecture, which is to serve as a foundation for building a product line. For physical products, product line architecture is the combination of subsystems, the interfaces between these subsystems, and the interfaces to external systems that collectively serve as the foundation for a stream of specific products (Meyer and Lehnerd, 1997). This is essentially a Deming or systems definition of architecture for product families (Deming, 1982).

Modular *product line architecture* defines the number and specific focus of core subsystems and interfaces, as well as the design criteria that will achieve robustness, scalability, and elegance in these subsystems and interfaces at both the data and programming level.

Architecture must also make a product line readily adaptable to new market applications by virtue of developing a new module or component and attaching it to others through predefined interfaces. Software system architectures that fail to incorporate extensibility doom the systems that are so designed to a short, chaotic and complex life. *In contrast, modular software product line architecture allows a system to evolve in an organised and efficient manner over time by making changes in any one subsystem transparent to other subsystems that already exist within the overall architecture*. Alternatively, if interfaces must be changed or be improved, or entirely new subsystems added, that work is isolated to a layer above the underlying core foundation subsystems within the architecture. Either way, making module improvements transparent to all other subsystems or isolating the impact they have to specific layers within the code reduce overall programming effort, while still providing users with all the new functionality offered by the improvements.

This is richly illustrated by the 'pipe and filter' architecture of the UNIX shell, where one can add a new programme to a command line without violating the functionality of commands that come before it. Alternatively, in Excel, one can continuously add 'macros' that perform certain special tasks but do not interfere in any way with the underlying input and output of the underlying spreadsheet. In Windows, there is the 'plug and play' PCI bus interface, which allows users to dynamically attach devices to computers and have them work automatically. These are simple yet powerful examples of the importance of clean interface architecture.

### *Software platforms*

*Platforms* are the actual subsystems and interfaces between the subsystems that are used by multiple applications. *The subsystems and interfaces, if used just once or in just one product, are not platforms*. The term platform demands reuse across several or more products. When a subsystem is shared or reused across several or more software

products, either within or across product lines, we consider it a product platform. Similarly, if several or more products use an interface among subsystems, it, too, is a product platform. In software, the platforms are modules of code that enable the development of end-user products, the integration of these products and other external products in the field, and the orderly evolution of those products over time to meet new customer needs.

The combination of product line architecture, focused subsystems, and robust interfaces between subsystems comprises what we like to consider the essence of a software company's bread-and-butter technology strategy.

### *Software applications*

These are the end-user products. Each product is a member of a software product line and each product uses the architecture as its structure of that software product line. Each software product uses the key subsystems and interfaces common to other members of the product line. Developers then create value-added modules to work in conjunction with these platforms to create new solutions. Or, as is often the case with software, end-users will customise certain modules on their own to achieve desired results.

Combining modular product line architecture with subsystem platforms and disciplined interfaces enables a company to create more readily a series of related software applications, each focused on a different use or purpose, potentially for different types of users. This is the first step on the path of mass customisation. The second step is the changes that users make to these software applications themselves to customise the software for their own respective purposes. Adhering to the layering principle, the goal is to allow the user to customise the software without corrupting the underlying foundation technologies – *e.g.*, the core subsystems and interfaces – that exist within the software application supplied to the user. Once again, the 'macro' developed by users of Excel is a perfect example. The statistical utilities that Microsoft has itself added to Excel comprise another clear example.

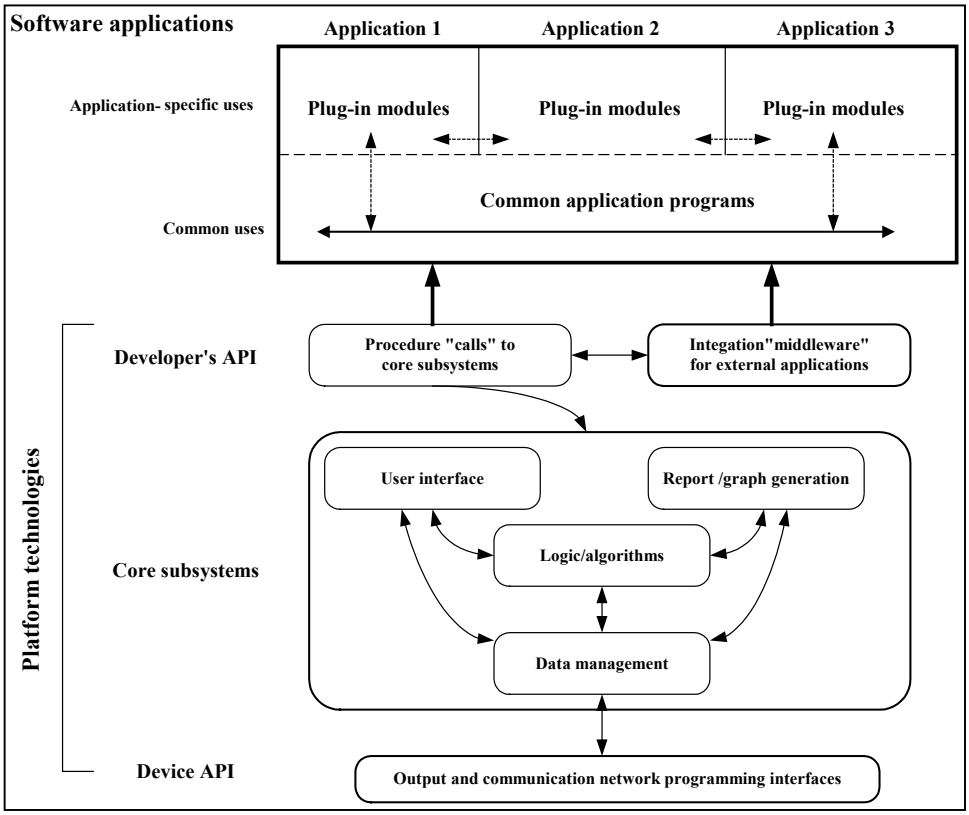
## 4 Creating a business strategy based on layered, modular architecture

Figure 3 presents a strategic management framework that integrates software product line architecture, software platforms, and software applications (Meyer and Seliger, 1998). The bottom half of the figure comprises the software platforms, *e.g.*, the key subsystems and interfaces that serve as the foundation for the product line. The product line architecture is the aggregate structure containing the specific number, purpose, and connections between these subsystem and interface platforms. For the purposes of discussion, we have shown a set of typical major subsystems often found in well-executed commercial software products:

- A major subsystem for developing user interfaces within applications, both for controlling programme executive, entering data, or requesting output. Within this subsystem are specific subsystems for calling graphical objects, designing menus, handling errors on the part of the user or the system itself, and accessing 'help.'
- A major subsystem for requesting data and organising it for reports or as data needed by other systems. This has traditionally been called report generation.

- A major subsystem containing for processing the logic in the system, be it algorithms for engineering-related calculations or financial calculations. These also often comprise libraries of specific sets of logic or functionality.
- A database management subsystem, containing methods for structuring, accessing, and combining data.

**Figure 3** Modular platform-based architecture and derivative software products




The core subsystems within the architecture can be powerful product platforms, the crown jewels of the software company. Sitting in a logical manner above these core subsystems in Figure 3 is a development environment specifically created by the firm to allow its own engineers, 3rd parties, and customers, to development software applications accessing and otherwise using the common subsystems and interfaces. This subsystem is known commonly as a Programming Application Interface, or API, and it is the basis for developing custom or specialised programmes based on standard software cores. Since the API is for other programmers, we refer to this set of interfaces as a Developer's API.

Interfaces deserve special attention in the case of software product lines. In fact, interfaces can be more important than any particular subsystem because they can allow the latter to be swapped in and out, or replaced with a newer version, at the need or convenience of the developer or the user:

- The interface between the end-user and the software applications. The benefits of a disciplined, often graphical interface here are well known and obvious. While approaches for good user interface design continue to evolve, simplifying complexity remains paramount. It is also interesting to see meta-interfaces emerge across different types of software applications, such as single sign-on interfaces in healthcare for patient safety and physician security.
- The interface between one software application and another software application, be it controlled by the user (as in a cut and paste between a word processor and a spreadsheet) or a real-time inter-process communication between two systems on the plant floor. Microsoft's Office is good example of the power of application integration at the user level – Excel spreadsheets are embedded easily into Word documents, to cite just one instance of this integration.
- The interface provided for developers of applications into the company's core subsystems. This is the Developer's API in Figure 1. SQL (structured query language) was a marvelous invention that allowed any programmer to define and access data from just about any relationship database management system. IBM, Microsoft, Oracle, BEA and IONA are competing heavily for next generation solutions in this area with their 'application services' middleware offerings.
- The interface between one core subsystem and other core subsystems. This is where code tends to get very messy and problematic for most software developers.
- Last, the interface between core subsystem and devices or networks. Traditionally, operating systems have provided these capabilities. In certain fields of embedded and other specialised computing, the software developer must build these interfaces itself. A storage management company for example, provides APIs to reach out across different types of networks and machines to gather storage information. Alternatively, a process control toolkit vendor provides interfaces to connect to and read data from different types of instruments and equipment on the plant floor.

One can categorise software interfaces as: a) those functions used by developers outside the company and b) those functions reserved only for internal use. Though the same architectural principles apply to the design of both types of interfaces, it is worth noting that interfaces exposed for external use are going to be much more difficult to change than interfaces used only within the company. The problem of *backwards compatibility* (how well does the current version of a product handle user customisations made to a previous version) is so difficult to solve that many software firms simply do not evolve their publicly exposed interfaces. Instead, many products support multiple versions of an interface. This provides perfect backwards compatibility, but is a maintenance and support nightmare, and greatly increases the complexity of the product. This is a persistent, widespread and intractable problem. Microsoft Windows is the classic example. Despite all the changes made in the last two decades, both to underlying hardware and the system-level APIs, Windows programmes written in 1984 will still run on today's Windows XP.

Also shown in Figure 3 is an Applications Integration Subsystem. This subsystem allows developers to integrate seamlessly a firm's applications with applications made by completely different software development entities. In a hospital, this might be a clinical information system whose control and logic integrate easily with the hospital's administrative and billing system, both made by completely different software vendors.

The top half of Figure 3 then comprises the software applications. Some of these applications are common to all of the target market applications chosen by the firm, and others are specific to each target use. This strategic framework helps explain the dynamic growth strategies seen in certain software companies where new software applications systematically leverage a company's foundation technologies, its major product line architectures, and its common subsystems and interfaces.

Further, if the marketing side of the enterprise has a well-managed business development function, a number of these domain-specific plug-in software applications can be developed by independent concerns other than the firm itself. These third party developers may then use the firm as a channel or license the firm's platform foundations and market their own software applications directly.

Figure 3 implies a variety of end user solutions for software products that can be for greater than that for convertional physical products. Kelly (1994) aptly described this industrial pattern as a 'swarm,' the evolution of an open universe of solutions around a robust software architecture – developed by one, modified by many, and used by many, many more. The trade-off is that the originated software company cannot possibly control all the developments and marketing activities of its independent partners, and neither should it try. The value derived from the product is a direct result of the connections formed among members of the community (Ongardanunkul, 2001). A user who develops his or her own plug-in modules for a major software product – be it Linux, Excel, or Oracle – can happily make those modules available to the larger community. This type of thriving software ecology can make the standard software product much more desirable to customers and therefore much more valuable to the company that produces it. It is the essence of mass customisation.

## 5 Application of these principles to achieve sustained growth: the MathWorks

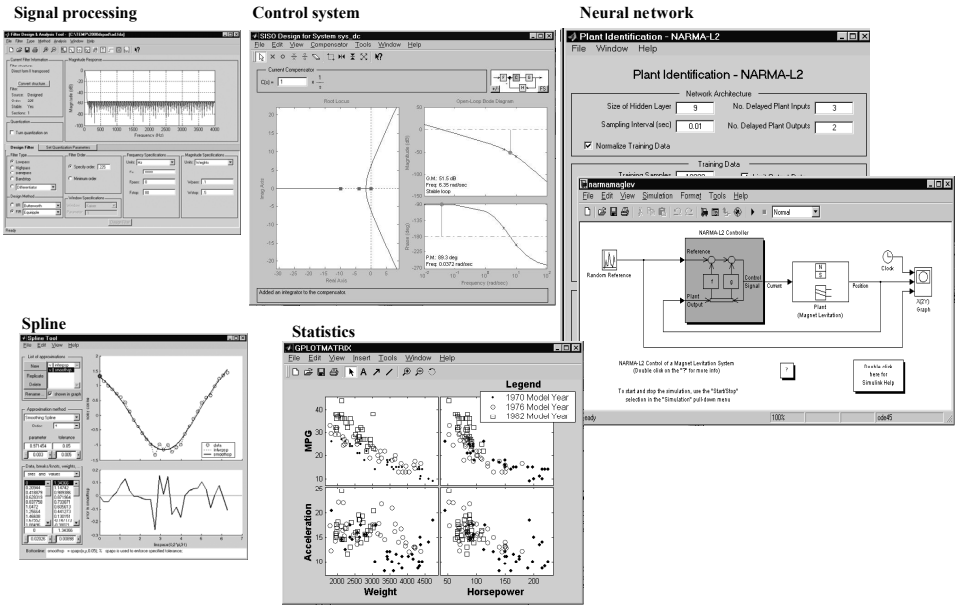
As we have indicated throughout, clean, robust interfaces allow a software product line to evolve gracefully, where better subsystems replace older ones, and where the software itself can be customised for a broad range of market applications. While these principles seem so basic, even the best of software developers tend to let interfaces to slip and slide, to multiply in number, and to conflict through overlapping scope or direct interference. This we shall see in the case of the MathWorks, a highly successful software company based in Natick, MA. The MathWorks has recently completed an enormous effort to create a more layered, modular architecture for its products to accommodate new market applications for its software.

The MathWorks' products are customisations of vertical market specific toolkits, where the toolkits themselves are developed on common language platforms. This creates a very basic yet fundamental hierarchy of technology layers: languages beget toolkits, that beget highly specific, personalised applications. These applications take the form of complex data analyses, visualisations, and system design tools for the engineering and scientific communities.

Figure 4 shows the examples of the types of applications and solutions that this company has created over time. The modelling of controls and digital signal processing systems are two key market applications for the company. There are many other

mathematical applications. Figure 4 also shows a neural network model for continuous learning and adjustment of a manufacturing process. The figure also shows some basic statistical functions. Over the past several years, The MathWorks has leveraged this technology into non-engineering markets, including the design of financial services products (modelling derivative instruments) and the modelling of systems biology.

**Figure 4** Matrix mathematics and statistical algorithms for different applications

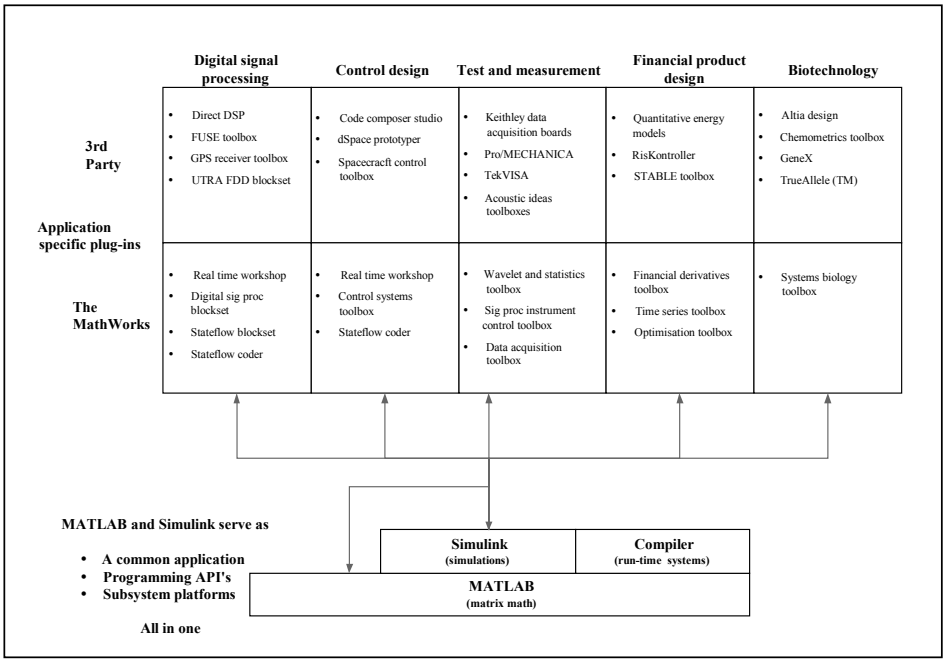



The company was founded in 1984 based on the premise that desktop computers would provide a viable platform for numerical computation. This vision was on target. Today, desktop computers are the primary platforms for mathematical modelling in corporations and professional organisations. In addition, the founders' warm spot for university learning made good business sense. Many engineers first learn how to use The MathWorks software as engineering students under highly favourable academic licensing arrangements and then wish to continue to use the software as professionals. This seeding of new users has been a powerful market development strategy.

More than a million customers worldwide now use the company's flagship MATLAB and Simulink products. Realistically, each one of the million customers has customised the software for his or her specific purposes. For example, each DNA sequencing algorithm created by bio-scientists is by definition different unless two scientists are working on exactly the same piece of DNA. Each financial derivative model is different, given the security or commodity and the macro and micro economic factors incorporated by the finance specialist into his or her model. Similarly, the automotive engineer will create a variety of simulation models for the performance of the different anti-lock braking systems that he or she has designed or procured for the various vehicles and models under development. In sum, The MathWorks' universe of applications is one of highly varied and deep complexity.

Figure 5 provides an overview of the The MathWorks' software product offerings. The company's products can be broadly divided into two categories: the language of technical computing and control design automation. A core product serves each one of these categories: MATLAB and Simulink. On top of these a series of application specific plug-ins, some made by The MathWorks itself, and others, by third parties.

**Figure 5** The MathWorks product strategy




## *5.1 The language of technical computing*

MATLAB is The MathWorks' flagship software product and is a language for mathematical modelling. To the user, it appears as an interactive environment that supports a scripting or programming language that is highly adept at creating, manipulating, and performing complex calculations on matrices of numbers. These scripts are contained in MATLAB *M-files* that are run typically just within the MATLAB environment. As another alternative, developers can combine these *M-files* with C/C++ programmes, processed by the MATLAB Compiler, and executed outside of MATLAB in binary form. About 20% of The MathWorks' customers use the MATLAB Compiler to share MATLAB-based applications with people who do not own MATLAB.

With MATLAB, the user can analyse vast arrays of data and produce graphics and charts, using the latest techniques for data visualisation. MATLAB's core functionality is extended via *toolboxes*, each consisting of a set of domain-specific functions, objects, and graphical user interfaces. For example, the Image Processing Toolbox adds image analysis (edge detection) and image processing (image registration) functionality to core MATLAB.

### *5.2 Control design*

MathWorks' second core product, Simulink, provides a graphical environment for the design and simulation of nonlinear dynamic systems. Simulink focuses on the control design automation market. Users construct their systems by building block diagrams, using the blocks provided with Simulink or those they have developed themselves. Extensions to Simulink's functionality are called *blocksets*, and like MATLAB's toolboxes, they enable users to solve more easily problems in a specific domain. The Neural Network graphic model shown in Figure 4 is an example of one of these blocksets. Or, the modules in the Communications Blockset can be assembled by the user to simulate the operation of a modem. Alternatively, there are blocksets for automotive applications. The anti-lock brake system is a good example, where the engineer wishes to simulate how the brakes react to different levels of weight and speed.

### *5.3 Mass customisation*

The MathWorks product line demonstrates how modular architecture enables mass customisation and drives commercial success. The core of the company's strategy is the MATLAB language. It provides both a foundation for mass customisation and a unifying framework that knits the MathWorks Toolboxes and Blocksets together with MATLAB and Simulink into a coherent whole. The MathWorks exploits modularity at two levels. First, at the language level, the division of sets of MATLAB functions into toolboxes allows customers to choose to pay for just those features their applications require. Second, at the language processing level, the modularity of the MATLAB Component Runtime enables the use of the MATLAB language in applications other than MATLAB itself, *e.g.*, programmers can integrate compiled binary MATLAB modules into other applications. MathWorks customers can invoke MATLAB functions as standalone programmes, from C or C++ development environments or from any application that supports Microsoft COM Objects or Microsoft Excel add-ins. The overall effect of this architecture enables MATLAB users to customise their MATLAB experience by adding a custom set of features to the MATLAB environment. This also applies to company's control design product, Simulink.

The flexibility in both feature set and operating environment has allowed The MathWorks to capitalise quickly on new markets. The range of problems that can be expressed in matrix or array form is vast and MATLAB's ability to provide solutions has been honed by years of mathematical research and algorithm design. To serve a new market, developers at The MathWorks customise MATLAB for that market by creating a new toolbox. With the wide range of capabilities in the base MATLAB product, the development of a new toolbox has proven far more rapid than writing an equivalent product from ground zero. Further, since the MATLAB language operates in multiple computing environments, including Windows and Linux, customers do not have to abandon their legacy systems to take advantage of MATLAB-based solutions.

One clear example of the flexibility to customise solutions for new market applications is the company's more recent expansion into the financial services market. The MathWorks' initial offering, the Financial Toolbox has grown to six toolboxes over just several years, and the revenue derived from these new solutions increased from tens of thousands to millions of dollars. Growth of this magnitude would have been impossible without the support of a truly modular architecture.

## 6 The MathWorks' architecture challenge and its solution

What would not be obvious to the casual outside observer is that by the turn of the millennium, MATLAB and Simulink had been heavily intertwined or co-mingled. The reason for this was that MATLAB was developed first (the first version shipped in 1984). When Simulink was developed in the early 1990s, it was based on the substantial amount of code and libraries that already existed in MATLAB. To achieve the types of functionality required for control design in highly complex automotive and aerospace applications, the company found itself having to add increasing amounts of code to MATLAB just to get the needed functionality in Simulink.

This intermingling of interfaces and the lack of clear layering caused problems for other development groups within the company. For example, the Compiler group was placed in a very difficult spot. Its role was to allow any user to create compiled versions of M-file programmes that could run in a standalone manner. Because MATLAB's architecture was not highly modular, the Compiler had to maintain a separate runtime library that mimicked the functionality in MATLAB. This made it nearly impossible for the Compiler team to keep up with all the changes made to MATLAB language by its own development team.

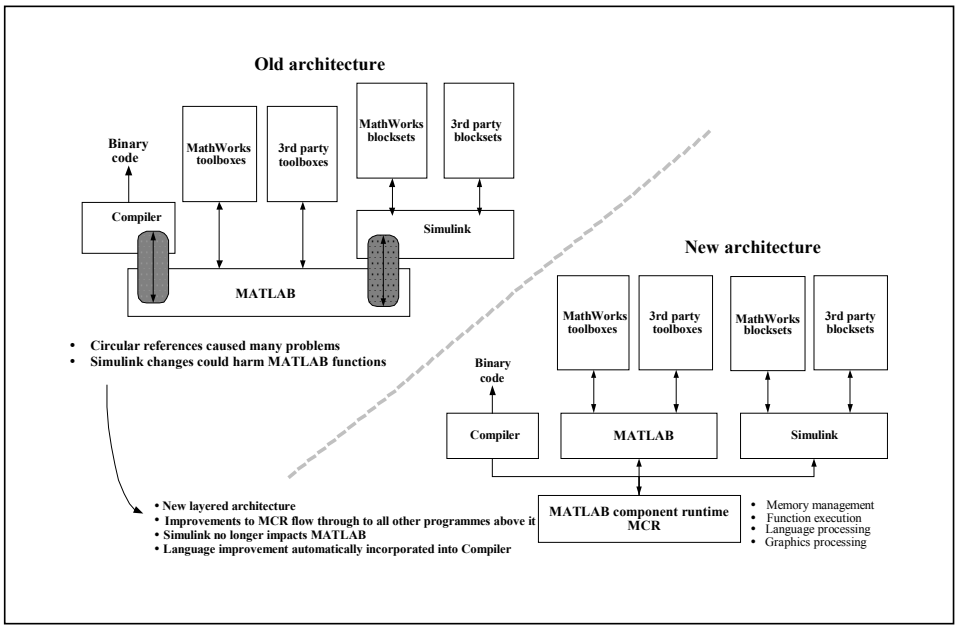
Other signs of unnecessary complexity existed in areas as basic as printing for MATLAB to print graphs, it had to load the Simulink libraries simply because the printing routines relied on certain key functions in those libraries. This established a circular dependency: Simulink required the MATLAB language, but at the same time, MATLAB required Simulink for certain key functions (such as printing). A change in Simulink could therefore break MATLAB. Circular dependencies such as this can be critical failure points in software.

In sum, The MathWorks could not enjoy the benefits normally associated with a layered architecture, where improvements in lower level modules help all higher-level modules, and bugs in those high-level modules that do not seep down into the lower level modules. Simulink and MATLAB were entangled, and Compiler users often had to wait several release periods to get access to the MATLAB interpreted M-file language improvements for their own respective run-time programmes.

In 2001, management decided to redesign the MATLAB-Simulink combination to create a new set of core software that would serve as a platform of shared components for MATLAB, Simulink, and the MATLAB Compiler. This new platform of shared components is called the MATLAB Component Runtime or MCR. This set of shared subsystems would form the core runtime environment for MATLAB, Simulink and the MATLAB Compiler. As one might expect, management formed and co-located a dedicated team that drew upon talents from the MATLAB and Simulink groups within the company.

Figure 6 shows the before and after architecture at its highest logical level. In this new architecture, The MathWorks' developers created a new layer of functionality to serve MATLAB, Simulink, and the Compiler. These shared platform components include what one might expect as common modules: memory management, function execution, language processing, and the creation of graphic objects.

**Figure 6** The further layering of the MathWorks architecture to increase modularity and flexibility for release R14 (2004)




There are numerous direct benefits to both users, and developers within The MathWorks itself. New functionality built into Simulink no longer weighs down MATLAB-only users, and visa versa. The Simulink team can add its own new code without worrying about ripple affects throughout MATLAB, (such as the printing function mentioned earlier). Further, improvements made to the language processor at the core platform level automatically flow through to the Compiler without additional programming. The Compiler development team no longer has to maintain its own standalone runtime library, vastly reducing the effort required to support all the features of the MATLAB language. Perhaps most importantly, the core MCR team (working on the bottom box in the new architecture shown in Figure 6) can be assured that the new functionality that it adds in areas such as language improvements, performance, and graphics processing will automatically flow through to MATLAB, Simulink, the Compiler, toolboxes, blocksets, and all the applications developed by users with them. The MCR has became a new and truly powerful product platform.

With this layered architecture as a foundation, the MathWorks and its customers have moved rapidly into new mathematical design and simulation applications. For example, end-users can more readily customise MATLAB and Simulink to suit their application requirements. Both programmes allow users to change the appearance and behaviour of the user interface and to add or modify core functionality. In MATLAB, for example, the user interface metaphor is the desktop – a working surface that holds all the tools needed for the day's tasks. MATLAB's default desktop configuration contains several of the most frequently used tasks, each contained in its own window. While users start with a pre-configured desktop, they soon proceed to create and save their own custom configurations. Users can further customise the desktop by specifying the fonts and

colours used to display text, adding new items to the main menu bar or shortcuts to the 'Start' menu.

Beyond these surface level customisations lies the customisation of core functionality: many of the mathematical functions in MATLAB are shipped with complete M-file source code, allowing users to copy or modify this code to create their own MATLAB-based solutions. MATLAB's open architecture also makes it easy for users to develop toolboxes of their own, as many have done. A MathWorks' customer often deploys these toolboxes internally in his or her department. However, a number of customers have added value to these toolboxes and now sell them as independent products. The MathWorks provides reference channel for numerous independent software companies that make plug-ins for MATLAB and Simulink, each with its own very specific mathematical functionality for specific applications.

In sum, the development of the MCR has facilitated greater end-user customisation by allowing users to deliver domain-specific MATLAB-based solutions to their customers. MATLAB users can select a set of MATLAB functions, knit them together with a customised GUI and use the MATLAB Compiler to deploy the final application. The deployed application contains only those functions required by the task and a more focused GUI than the general-purpose MATLAB desktop. All this customisation would not be possible without the newly architectured MATLAB, Simulink, and Compiler, as well as the MCR layer developed underneath and shared by them.

In other words, The MathWorks architecture-layering activities have made mass customisation with its software even more feasible, and less painful, than ever before.

## 7 Concluding remarks

Even with such examples, our observation is that many practitioners, engineers as well as business managers, do not really understand how good architecture in software drives business growth for systems-developing companies. Poor architecture leads to problems for developers and users alike. In software, this means ineffective project planning and execution, as well as a myriad of bugs for new software brought to market. Developers suffer; users suffer more.

This observation is not ours alone. One firm that studied US commercial software projects in 2002 found such poor planning that companies cancelled about a quarter of their projects outright with no final product, costing the developing companies $67 billion (Mann, 2002). Overruns on other projects cost another $21 billion. Poor planning and cost overruns are clear indicators of the absence of robust architecture and product line strategy. The same study found that 80% of the budgets for software projects were often devoted to repairing bugs *prior to commercial release*. Bug-fixing after commercial release was an additional and substantial cost.

These are sobering data. The hypercompetitiveness of the software industry forces many software firms to rush new features out to their users. Firms spend insufficient time developing and enhancing the underlying architectural foundations required to support error-free, secure programmes. As software firms grow, and add more programmers to development teams, development projects often become encumbered in poor communications, leaving even less time for thoughtful consideration of layered, modular architecture (Brooks, 1995).

Perhaps there is no better example of complexity and renewal in software, and none with greater ramifications, than Microsoft itself. It can be argued that Microsoft's lack of a layered, modular architecture opened a window of market opportunity for Linux. Microsoft's architectural deficiencies have made the addition of new functionality not only expensive but also error-prone. As Microsoft has added considerable functionality with each new version of its Windows operating system, that functionality has carried with it a substantial number of new bugs with each new release. The very day that Microsoft released XP, the company posted 18 megabytes of bug fixes, security patches, and other updates for XP on its website (Mann, 2002). Another industry observer reported that that Windows NT 4.0 had 10,000 known bugs upon commercial release; Windows 2000, which followed NT, 63,000 bugs upon commercial release; and Windows XP, released in 2001, over 100,000 bugs (Rodrigues, 2001). Bill Gates testified during the Microsoft antitrust trial that Windows would not function if customers removed individual modules such as the Internet Explorer browser or the Exchange e-mail programme (Mann, 2002,p.36), a clear sign of entanglement between modules.

Like The MathWorks, Microsoft decided to confront its problems by creating a new architecture, which it calls .Net. In Microsoft's new architecture, one finds a modular, layered programming environment with highly structured interfaces that serves as an applications programming interface pervasive across Microsoft's database and programming tools and libraries. It has more platform capabilities: .Net provides a common foundation for memory management, function execution, language processing, and a common execution environment for all of Microsoft's programming languages (such as Visual Basic or C#). It also features a highly specific and visible layer for security. In short, .Net represents a strategic investment in modular, layered architecture by enhance the productivity of its developers and users the world's largest software company.

The good news is that an increasing number of major software companies are now applying the principles described in this article in ways similar to The MathWorks and Microsoft. They realise that effective standard product development and follow-on user customisation of these standard products relies on layered, modular architecture. The business benefits of undertaking architectural renewal of software include:

- The flexibility for users to customise various parts of the software for their own specific purposes, thereby making the original software both more useful and longer-lived.
- Faster time to market for new versions of the software. With well-structured interfaces, the product is less complex and integration, which is the hardest area in software engineering, becomes manageable.
- Greater reuse of code for different versions or market applications of the software.
- Improved risk management. If one major subsystem or component of the software has problems, the focus of the corrections or bug-fixing can be localised more readily to that subsystem. Further, developers can rewrite the module without affecting the other pieces, vastly simplifying not only the programming but also the testing effort.
- The potential for rapid third party plug-in development, with the potential for new market applications outside the original software company's business plan.

It is these business benefits that the software developer must understand and articulate before embarking on the type of architectural renewal described in these pages, renewal that is the foundation for effective mass customisation.

## References

Bass, L., Clements, P. and Kaxman, R. (1998) *Software Architecture in Practice*, Reading, MA: Addison-Wesley.

Brooks, F. (1995) *The Mythical Man-Month – Essays on Software Engineering*, 20th Anniversary Edition, Reading, Massachusetts: Addison-Wesley.

Clements, P. and Northrop, L. (1996) 'Software architecture: an executive overview', *Technical Report CMU/SEI-96-TR-003*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Deming, E. (1982) *Out of Crisis*, Cambridge, MA: MIT Press.

Dilip, S., Nord, R. and Hofmeister, C. (1995) 'Software architecture in industrial applications', *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington: ACM Press, 24–28 April, pp.196–207.

Von Hippel, E. (1988) *The Sources of Innovation*, Oxford, England: Oxford University Press.

Jazayeri, M., Ran, A. and van der Linden, F. (2000) *Software Architecture for Product Families*, Reading MA: Addison Wesley.

Kelly, K. (1994) *Out of Control*, Reading, MA: Addison-Wesley.

Manis, R. and Meyer, M.H. (1986) *The Unix Shell Programming Language*, Indianapolis, IN: Howard Sams & Company.

Mann, C. (2002) 'Why software is so bad', *Technology Review*, July–August, Vol. 105, No. 6, pp.32–38.

Meyer, B. (1992) *Eiffel: The Language*, Englewood Cliffs, NJ: Prentice Hall.

Meyer, M.H. and Lehnerd, A. (2004) 'Modular platforms and innovation strategy', in R. Katz (Ed.) *The Human Side of Managing Technological Innovation*, New York. N.Y.: Oxford University Press.

Meyer, M.H. and Lehnerd, P. (1997) *The Power of Product Platforms*, New York, N.Y.: The Free Press.

Meyer, M.H. and Seliger, R. (1998) 'Product platforms in software development', *Sloan Management Review*, Vol. 40, No. 1, pp.61–74.

Ongardanunkul, J. (2001) *Introducing Products with Network Externalities*, Department of Economics, Boston College, November.

Park, S. (2001) *Integration Between Hardware and Software Producers in the Presence of Network Externalities*, Department of Economics, SUNY Stony Brook.

Pine, J. (1992) *Mass Customization: The New Frontier in Business Competition*, Allston, MA: Harvard Business Press.

Rodrigues, P. (2001) 'Windows XP Beta 02: Only 106,000 bugs!', *LowEnd Mac*, 8 March.

Sharman, D. and Ali, Y. (2004) 'Characterizing complex product architectures', *Systems Engineering Journal*, Vol. 7, No. 1, pp.39–44.

Simpson, T., Umapathy, K., Nanda, J., Halbe, S. and Hodge, B. (2003) 'Development of a framework for web-based product platform customization', *Journal of Computing and Information Science in Engineering*, Vol. 3, Iss. No. 2, pp.119–129.

Soni, D., Nord, R. and Hofmeister, C. (1995) 'Software architecture in industrial applications', *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington: ACM Press, 24–28 April, pp.196–207.

Steward, D. (1981) 'The design structure system: a method for managing the design of complex systems', *IEEE Transactions on Engineering Management*, Vol. 28, pp.71–74.

Sundgren, N. (1995) 'Introducing interface management in new product family development', *Journal of Product Innovation Management*, Vol. 16, pp.40–51.

## Note

1 The join command combines two databases on common fields. The fourth column of the result contains the numerical amount of the purchases. The sort command sorts on this column, in reverse order. The tee command splits the output to a file, and into the mail programme. Everything is linked through pipes, and all command produce ASCII delimited output and expect that as input.