6 SURF: ACHIEVING QUALITY THROUGH SOFTWARE REUSE—A PROCESS IMPROVEMENT EXPERIMENT IN IBM ITALIA

Marco Riva
Alessandro Agostoni
IBM Italy

Abstract

Managing software quality in a technology-driven, medium or small size development organization, characterized by a very heterogeneous environment, is a complex task that requires particular attention in order to be effective. This paper describes a process improvement experiment (PIE) targeted to the introduction of software reuse practices as an approach to software quality. Main objectives of this PIE have been the reduction of the development cost, of the cycle time, and of the maintenance effort. The most innovative aspect has been the introduction of a "layered" framework for the production of the reusable code. The framework proposes a decomposition of an application into five distinct functional layers and has been designed to maximize reusability when a large number of different hardware and software target platforms must be addressed and the product line is not homogeneous and stable over time. Organizational, technical, and cultural changes have been monitored during the PIE time frame. Surf is the 18 months ESSI PIE n.23752 started on May 1, 1997.

Keywords: Software reuse, software quality, software metrics, PIE, RMM.

A Look Inside Quality

If you look for the word *quality* in a dictionary, you will find it defined as *degree of excellence*. As we will see in this paper, evaluating (and improving) the quality of a software product is a very complex task, which does not fit in that simple, three word definition.

In the literature, there are many different definitions for the quality of an artifact. For example, Garvin (1984) gives five different perspectives on quality: the *product* quality, based on quantitative measures of a specific product's parameters; the *manufacturing* quality, based on the definition of the manufacturing process and monitored with inspections and statistical analysis; the *user* quality, based on the qualitative perception of the user (the customer) and focused on the satisfaction of the user's requirements; the *value* quality, based on a mediation between project's constraints, such as development cost, time, and target domain; the *transcendent* quality, based on user's personal perception and affective feelings for a product.

From our point of view, to better understand the problems related to software quality evaluation, we can try to simplify the approach to quality, limiting the scope to the three main viewpoints: the user, the process, and the product.

Probably the most evident aspect of software quality is the one perceived by the customer. This kind of quality is usually referred as *external* or *observed* quality. The success of a product can heavily depend on the quality level that it shows to the final users. This includes aspects such as satisfaction of the user's functional requirements, as well as execution speed, ergonomic aspects (ease of use, look, and feel), and robustness. Directly addressing this aspect of quality goes beyond the scope of the PIE. A bottom-up approach has been adopted, assuming observed quality being a logical consequence of a well defined development process and a robust product architecture. Looking from the development process perspective, the quality of a software product is higher when the process that generates it is well defined, stable and completely under control. Quality is achieved with inspections and statistical analysis during the development phases. The ISO9000 and the CMM (capability maturity model) are the most well-known examples of this approach. The aspect of process qualification has been addressed with specific actions external to the PIE. Otherwise, the PIE action has been fundamental for the definition and consolidation of the software development process.

Ultimately, however, the quality of a software asset can be considered an intimate characteristic of the software itself. This is the case in the product-based view of software quality. This kind of quality, also defined as *intrinsic* quality, has been considered the key factor when defining the PIE implementation strategies. Each action in the process area has been considered with maximization of intrinsic quality as the main target. All other objectives (reduction of the development cost, reduction of the cycle time, reduction of the maintenance effort) have been chosen as the measurable consequences of an *intrinsically good* design.

Reuse practices have been identified as being potentially the best way to capitalize on intrinsic product quality. The initially greater effort needed to develop reusable components, having intrinsic quality as their foundation, should translate into progressively increasing savings in the medium and long term. This has been the PIE's challenge.

Experiment Scenario

The organization involved in this process improvement experiment is an IBM Italia department, Manufacturing Applications and Industry Solutions Development (MA&ISD). MA&ISD is responsible for the development of a wide range of high technology electronic products, including hardware, firmware, and software. MA&ISD has a grid organization, with vertical departments having specific responsibility (development programs, system product development, software development, quality and methodology) and horizontal task-based development teams. Project managers play an important role in resource allocation and communication within the vertical areas. This organizational schema, summarized in Figure 1, is giving good results in terms of flexibility and efficiency.

The MA&ISD organization contains about 60 people, almost equally located within the vertical areas. Each new project is assigned to a team of people taken from vertical departments. The composition of a team depends on the project requirements and on the time constraints. The average size of a team is about seven people, but critical projects can require a greater effort (up to 15 pr 20 people). Due to the MA&ISD product typology, a team is generally composed only partially of software designers. The head count specific for software development in the average project is about three people, covering about 40% of the total design effort. The Surf PIE has been implemented in the software department, namely ESD (Engineering Software Development, see Figure 1). ESD contains 15 people.

MA&ISD is a development department. Its activity is historically related with the products of the Italian IBM manufacturing sites. Over the last year, the role of MA&ISD evolved toward a more mature stage. Today, it is acting as a development laboratory and consultant for several IBM worldwide locations and external customers in Italy and Europe. Its products portfolio is very heterogeneous: MA&ISD is a task driven organization rather a product line based organization. The main areas in which its products can be located are system integration, data security (cryptography), multimedia, and communication. Having such a wide activity spectrum makes it very critical to manage resource allocation in an effective way. This has been the main reason for the creation of the grid-shaped organizational structure. As soon as a department has been officially established to manage software development activities (this happened in the

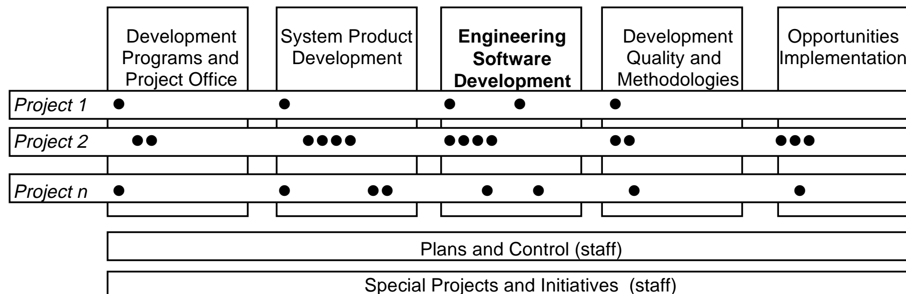

Figure 1. MA&ISD Organization

early 1990s), software itself began to be treated as an independent product and action started to focus on software process improvement.

The software development process before this PIE can be modeled basically with a classic waterfall model, characterized by a sequence of activities: requirements collection, analysis, design, coding, validation, and maintenance. In the last years, due to the experimental nature of several projects, in which requirements were not stable but were changing during the whole project time frame, some adaptations were made to the classic waterfall model, mostly enhancing the feedback loops between each step of the process. Another stimulus toward a modification of the process model came from the progressive introduction of the object oriented techniques inside the organization. Object orientation and rapid application development techniques does not fit exactly with the waterfall model. The Surf project helped to achieve a further refinement of the reference model for the software design process. Anyway, the waterfall model can be considered a good approximation for the typical life cycle before the PIE.

People in the MA&ISD organization have generally high specific skills and a background of work experience in several international projects in other IBM locations (in Europe, USA, and Japan). From the software point of view, a broad skill spectrum is covered, from low-level languages (machine code, assembler), to C, C++, Smalltalk and Java. Deep knowledge on many platforms, such as the X86 family, the PowerPc™, and embedded systems is also part of the MA&ISD skill set. Well experienced designers work in the firmware as well as in the device driver areas. The software department also has relevant development experience with visual building tools such as IBM VisualAge. Most of the designers have a good knowledge of the object oriented techniques.

The Baseline Project

Surf used the Datavideo and Datasat projects as a baseline for implementation and measurements. Datavideo is the result of a partnership between IBM and RAI (Radiotelevisione Italiana) to develop a complete end-to-end solution (hardware, software, services) to broadcast data to a high number of receiving stations using the spare capacity of the television signal. Digitally encoded data are inserted in the VBI (vertical blanking interval) of the analog TV signal using the base concept of the Teletext but with a service targeted to professional market sectors. With Datavideo, the same data is transmitted simultaneously, on a subscription base, to a wide community of users geographically distributed. Datavideo is used in various market sectors: stock exchange data, software distribution and upgrade, professional categories such as notaries and lawyers. Datavideo is a unidirectional data transmission network with three nodes: *information providers*, that are the owners of the data to be diffused, the *carrier*, that provides the transmission infrastructures, and the *end-users*. Every end-user receiving station is made up of a personal computer equipped with a special card and connected to a conventional TV antenna. Relevant functionality is implemented in software.

The main characteristics of Datavideo software are about 150K lines of code, implementation of three layers of the ISO-OSI protocol stack, mapping on three different software platforms (DOS, OS/2, and MS Windows), two major software releases, and a high number of installed modules (now about 30,000). The Datavideo project started

in late 1989. Analysis, design, and development activities were carried out through 1993, when the product was launched on the market.

The total Datavideo development effort (including hardware, software, and services) can be estimated in about 2.5 million ECU's (30 man years of effort).

In 1996, a major change of the data error recovery policy at the file level caused an important review of the existing code with relevant software redesign and development activities.

Due to the technology evolution in broadcast data transmission using the TV signal, a follow-on of the Datavideo project, namely Datasat, started in 1996. Two major releases of Datasat are planned.

Although the transmission media is different (digital satellite TV versus analog terrestrial TV), this new project maintains a high degree of functional commonalty with Datavideo, at least in the higher layers of the protocol stack. At level three of the ISO-OSI stack (network layers), a higher commonalty, around 70%, can be found, including logical data channels multiplexing, context dispatching policies, single and group addressing mechanisms, connection oriented and connectionless services, and time services. About 80% to 90% in functional commonalty can be found at the transport layer. Message transfer protocol (MTP) and data transfer protocol (DTP) can almost be ported "as is" from Datavideo to Datasat.

Establishing the Culture of Quality

In order to define an effective strategy for a reuse based software quality improvement, an analysis of the existing environment has been carried out to identify the key actions for the PIE implementation. The main path for the PIE has been identified as:

- definition of a set of indicators to keep process and product parameters under control;
- introduction of a new reuse-oriented cultural environment;
- technical definition and implementation of a reuse-based design framework tailored to the organization's needs.

In addition, three main assumptions have been made to guide the subsequent activities:

- *Holistic approach:* any action taken in the cultural, process, or product area should take into account mutual interactions with other areas. It should also integrate with other improvement actions already in place or planned inside the organization, like the ISO9000 certification, the introduction of common tools, etc. This is to eliminate redundancies and to make actions more effective.
- *Gradual introduction of changes:* every change in the development process and new tools introduction must be fine-tuned in order to minimize the additional workload on the involved people. This is to make the acceptance of changes easier for people.
- *Specific tools:* the adoption of a proper set of tools has been recognized as a key for the maximization of the potential benefits. Tools selection and their introduction in the organization must be planned accordingly with the two preceding assumptions and with the whole organization's plans and budgets.

productivity = Functionality/Person days
lead time = Calendar time/Functionality
quality = Reported errors/Functionality

Figure 2. A Simple Set of Process Metrics

Defining Metrics

The first step has been the assessment of the initial situation. Some fundamental indicators have been set to keep the process improvements under control. The parameters chosen were *productivity*, *lead time,* and *quality*, measured as described in Figure 2.

Note that functionality has been calculated starting from function points and lines of code, and errors have been weighted based on their gravity and considered for any mayor release life time. Expected benefits for the baseline project have been defined as 25% increase in productivity, 30% reduction in time to market, and 25% reduction in maintenance effort. In this phase, simple indicators were preferred. A more comprehensive set of process metrics, taking care of the business aspects in more detail, is expected to be refined after the completion of the Surf project.

As far as *product* metrics, the FCM *(factor – criteria – metrics)* model, proposed by REBOOT (Karlsson 1995, pp. 129-162), has been adopted. The original model has been tailored to the organization. The defined model, based on two main attributes, *quality* and *reusability*, is summarized in Figure 3.

A simple set of metrics has been defined for each criteria. The initial choice, according to the assumption made about gradual introduction of changes, has been to limit the number and complexity of product metrics to a minimal significant subset of the one suggested by REBOOT (Karlsson 1995, pp. 113-162).

In order to minimize the differences in the measures due to different coding styles, a change was made in the calculation of the LOC (lines of code) and COM (lines of comment). The new indicators have been defined as *normalized LOC* and *normalized COM*, as explained in Figure 4.

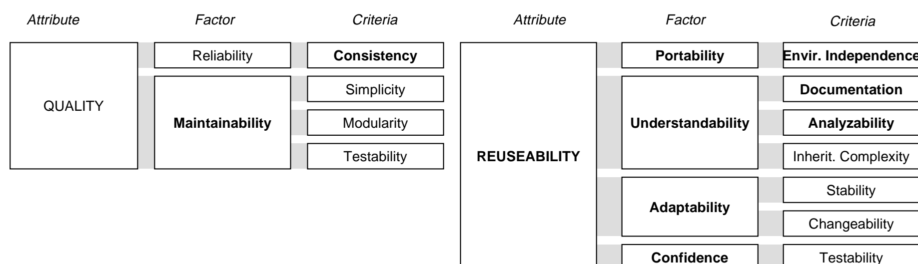

Figure 3. FCM Model for the Surf Project

nLOC = number of characters of code/reference line length
nCOM = number of characters of comment/reference line length

Figure 4. Normalized Metrics for Source Code

The reference line length, representing an hypothetical average line, was fixed at 30 characters. The number of characters of comments also include the *in line* comments (single line comments following a line of code) that are usually not counted by available tools. A tool has been written (in Java) to perform this calculation on source files.

Some commercially available tools for source level analysis were evaluated. In general, they were found to be too complex to manage and too expensive to maintain. In addition, none of them were able to cover all of the different languages and development platforms of the organization. Other required product metrics, like the McCabe's cyclomatic complexity, the inheritance depth, fan-in, fan-out, and others, have been extracted using a public domain tool, CCCC (C and C++ Code Counter), that has been slightly adapted from its original version. To help designers in the interpretation of product metrics, a tool has been created to display the data in the FCM (factor criteria metric) model in a graphic format. The tool, implemented as a distributed Java application, reads from the repository the numeric data associated with reusable assets and converts them into Kiviat (radar) diagrams. Factors, criteria, and metric values are normalized in the range 0 to 1 (with 0 as worst and 1 as best) prior to being visualized.

The specific problem of the reuse maturity has been addressed with the RMM (Reuse maturity model), a questionnaire defined by the REBOOT project (Karlsson 1995, pp. 201-215), the purpose of which is to identify an organisation's position with respect to software reuse maturity. The RMM model identifies five key areas: oOrganization, project management, assets classification, metrics, and development process. For each key area, a specific set of *key factors* is then considered. The RMM assessment has been conducted in the form of self assessment, involving a manager, two designers, and an external consultant.

The most critical areas evidenced by the initial assessment have been the organization (with respect to reuse planning, cost/pricing, and legal issues), the reusable assets management (the software assets were not stored in a reusable way before the PIE), and the process and product metrics (while some control over certain process variables was in place, reuse oriented product metrics were totally absent).

Improving the Culture

The first action in the Surf project has been to establish a good level of awareness about software reuse principles inside the organization. The choice was to use an "involvement oriented," rather than a more simplistic "information oriented," approach. People have been involved in the new design framework definition, rather than just informed about it. People interested in the Surf project have been able to follow its evolution through a Surf intranet page. This approach gave very satisfactory results, allowing rapid improvements to the framework and establishment of a more robust "quality culture"

inside the whole organization. A general introductory workshop as well as various presentations targeted to the people involved in "for reuse" and "with reuse" activities have been held.

The initial assessment made evident the lack of a common way to exchange design data between the people inside the organization. An adequate information exchange level has been recognized as an inalienable need in order to make reuse introduction effective. To address this problem, UML (unified modeling language) has been chosen as the standard way to document projects. UML is a visual modeling language for specifying, visualizing, constructing, and documenting software systems, developed by Grady Booch, Ivar Jacobson and Jim Rumbaugh. The current release (1.1) is the result of joint efforts of the original team and several companies that found UML to be strategic (HP, IBM, Microsoft, Oracle, Rational Software, and others). A synthetic but comprehensive view of UML is given by Fowler and Kendall (1997). The main reason that lead to the choice of UML is that it is presented as a language (not a methodology). This approach gives the designer an effective, standardized way to visualize the models as well as the freedom to follow a favorite design methodology.

Other actions have been started to improve the culture of the software development area, mainly focused on object oriented techniques, rapid application development, and process standardization. The defined training activities, summarized in Figure 5, have not been considered as isolated events, but coherently, with a *holistic* assumption. They have been organized in a comprehensive education tree, focused on both managerial and technical aspects, the purpose of which is the integration of the software planning, management, and design activities.

The educational path shown has not been fully exploited inside the PIE; it has been integrated with other MA&ISD initiatives.

Managing Reusable Assets: The Tools

A tool for the management of reusable assets has been considered a key factor for reuse success in the medium and long term. EuroNotes, developed by TXT (the Surf partner), has been chosen for this purpose. EuroNotes is a software platform targeted at exploiting World Wide Web client-server architectures for remote access support and information

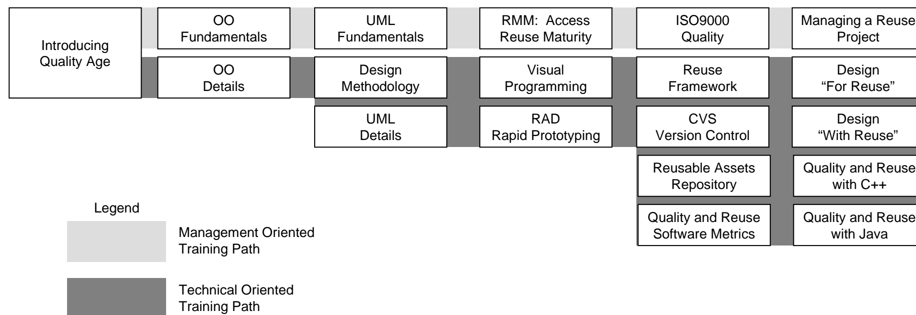

Figure 5. Surf Training Paths

management. It allows the user to insert, search in a structured way, and retrieve assets from any workstation, providing it is connected to the Internet or an Intranet.

The remote access capabilities of EuroNotes make it suitable for the widest accessibility, enhancing the business opportunities related to software reuse. One of the most interesting features of EuroNotes is the asset subscription capability. A user can subscribe to an asset: the repository sends an e-mail message to the asset owner notifying the registration of the new user, and sends e-mail notifications to the subscriber as soon as a change is made to the asset (new releases, new documentation or samples, etc.). EuroNotes also has error logging capabilities and a discussion forum can be associated directly with any asset. The EuroNotes data model is very flexible and it has been tailored to fit the layered framework architecture. Special views have been defined to allow different users easy access to the different typologies of reusable assets. The repository can be browsed by layer, by project, or by domain, and several search mechanisms allow the users to find the desired assets in an effective way. In addition, the EuroNotes tool, based on Lotus Notes/Domino, perfectly integrates with the existing IBM office automation environment, already based on Lotus Notes.

Future plans include full integration of Euroware with a configuration management system and implementation of an automatic system for automatic accounting of reuse activities.

Adapting the Process

To support the integration of the new activities in the development process, a new professional figure has been defined: the *reuse manager*. The main role of this manager has been the coordination of the reuse activities, such as organizing brainstorming sessions for cross project reusability evaluation, performing training and personal tutoring about the reuse practices and the design framework, and administering the reuse repository.

The existing life cycle model has been analyzed and corrected to better fit into the reuse oriented development schema. Following the *gradual introduction of changes* principle, the original waterfall model has been adapted rather than radically changed. The introduction of reuse practices has been achieved with two different actions, focused on the two fundamental aspects: “with reuse” and “for reuse” activities.

find similar requirements	domain analysis	find comps	understand		
Requirements Collection	**Analysis**	**Design**	**Coding**	**Validation**	**Maintenance**
reuse requirements	reuse analysis	reuse design	reuse components	already done (externally)	already done (externally)

Figure 6. The Integration of Reuse Activity in the Development Life Cycle

The "with reuse" activities have been integrated in the design flow. They imply both additional costs and benefits, as evidenced in Figure 6, where activities generating additional costs have been placed on top and those generating savings have been placed at the bottom of the original activity.

The requirement collection phase has been integrated with a specific session targeted to verify the requirements against similar projects in the same domains and also with other domain, based on potential "reuse affinity." This activity requires an additional effort, but savings can be obtained by reusing requirements already defined in the domain. In any case, the quality of the collected requirements benefits from existing experience.

The analysis phase has been evidenced as the key phase in the reuse oriented design process. The usual analysis phase has been enforced, including dedicated sessions targeted to cross project reusability evaluation and potential reusable subsystems identification. Particular attention has been paid to the identification of potential reusable subsystems, in the same domain (specific components) and across different product typologies (general use components). Object oriented techniques have been confirmed as the best way to implement reuse oriented analysis. In this phase, the reuse manager should verify the compliance of the architecture under development with the layered design framework.

The design phase has been integrated with a phase of retrieval and evaluation of reusable components. This activity is based on the functionality of the EuroNotes repository. Because the repository also holds design information, some savings can be achieved here by reusing existing design information from other projects in the same domain.

In the coding phase, the major overhead is the time needed to understand how the retrieved reusable components work. If the components retrieved from the repository have been well documented (this means both complete documentation and a standard format), the coding time saved is expected to be considerably greater than the understanding time.

The validation phase is the one that mostly benefits from reuse in terms of quality. Reused assets do not require any further validation. Validation of reusable assets is done once and benefits increase whenever the asset is reused.

The maintenance phase has been simplified, at least in terms of the project's scope. In fact, maintenance of reusable assets is performed under another project's responsibility. Again, maintenance of reusable assets is done once, and every project that reuses an asset acts as a benefit multiplier.

Note that an additional cost has been incurred training the designers and setting up the reuse oriented mind-set in the development team. This effort has already been evidenced in the previous paragraph, and is not shown here because it is considered as external to the product's life cycle.

"For Reuse" Approach: The Promotion Mechanism

One of the commonly recognized obstacles to the introduction of reuse practices is the additional effort required to produce reusable assets. The production of a reusable asset implies several new activities, such as a more accurate and complex requirements

collection and analysis phase. Again, the development phase must take care of needs that came from other projects. Validation of produced assets assumes a greater relevance, extending the designer's responsibility beyond the single project's scope. All of these new activities can create unaffordable problems when the project has strict time constraints. The risks are the creation of an obstacle for the acceptance of reuse principles by the designers and masking potential benefits to the eyes of the management.

To overcome these problems, some of the "for reuse" activities have not been fully integrated in the design flow. Instead, the "for reuse" implementation strategy has been defined with a mechanism of *promotion*. The activities strictly related with reuse, such as specific documentation (creation of the reuse guide and reuse tutorials for the reusable asset), code validation, and proper integration in the reuse repository, have been kept out of the single project's life cycle. They have been considered part of an external project, called the "base project." This approach has two main advantages: first, it allows concentration of effort of the single project on the specific targets, allowing reduction of overload due to "for reuse" activities; second, it allows better identification and measurement of the reuse activities, making monitoring of the reuse costs and benefits easier. This is because most of the activities that are specifically reuse oriented are grouped inside the same project.

The effectiveness of the promotion of reusable assets heavily depends on the right implementation of the design rules defined by the layered architecture. As we will see in the next section in greater detail, the layered framework itself leads to the design of potentially reusable assets. Following this design schema, the major overhead on a project affects the initial phase (requirements collection and analysis) that is commonly the less critical, in terms of time constraints, in the whole project life cycle. In addition, a more accurate analysis phase is expected to generate a more robust and stable design, repaying the additional effort with a better final product quality.

During the Surf project time frame, these activities have been performed by the reuse manager. Depending on the success of reuse introduction, a specific team could be created to manage promotion and "for reuse" activities in general.

Implementing Reuse: The "Fade to Gray" Approach

From the product point of view, the analysis and redesign of the Datavideo software allowed definition of a structured design framework. The framework structure originated through a set of needs that became evident during the Surf meetings and discussions. These needs can be summarized as follows:

- During the lifetime of a software product, it may happen that the adapter (or some hardware, in general) changes several times, requiring adaptations in the software that controls it.
- During the lifetime of a software product, it may happen that the functionality of the solution changes several times. Most of the time, new functionality is required.
- During the lifetime of a software product, it may happen that the technology changes and the product will have to be implemented in a extremely reduced time frame using new languages, models, or platforms.

These basic needs put in evidence the weak points of a monolithic application. Previous experience already made these weakness apparent, but a common solution strategy had never been implemented before the Surf project. Some ad-hoc solutions were adopted in the past to solve these problems. The reuse-oriented approach acted as a positive stimulus toward a more accurate and effective definition of a general software architecture. The problem has been analyzed at two distinct levels:

- *At the application level,* to define the right functionality splitting within a single application. Such an architecture allows localized changes due to specific needs into appropriate subsystems. This is in order to minimize redesign efforts due to changes in the application domain, such as new hardware, new functionality, new target platforms, etc.
- *At the domain level*, to identify commonalties between different products in order to maximize reuse effectiveness across projects and improve skill allocation effectiveness.

The Layered Framework

A great effort has been put into analysis of past design experiences, with particular attention to the Surf baseline project, Datavideo. The analysis evidenced the fact that often the only way to effectively implement a change (even a small one) in the functionality of a software solution has been to assign the task to the designer that originally developed the application. itself. Other solutions, involving people not directly involved in the original project, caused a considerable increase in the lead time. There are two primary reasons for this problem:

- *Lack of documentation:* the design specific documentation was not sufficiently complete, generally with respect to the architectural description.
- *Lack of standards:* every project was documented in a way that reflected the single designer's preferences, without a predefined standard. This occurred with both the description languages (natural language, visual modeling, etc.) and the tools (document format, word processing programs, etc.).

The PIE addressed both of these problems by suggesting a way to split a complete solution into a layered application, defining standard interfaces between these layers, in order to make the adaptation activity easier to be implemented.

The by-layer split of an application into distinct parts has several advantages:

- When a change is needed, its scope is limited to the interested layer, and does not span the whole application, making fixes easier to implement.
- The standard design and documenting architecture for all of the components developed inside the same layer makes the design easier to understand. This allows assignment of responsibility for changes to people other than the original designer with minimal additional effort.
- The separation between layers allows a designer with specific knowledge in one layer to act "safely" with assets inside that layer, without the need to know anything about the implementation details of other layers.

The proposed solution identifies five different layers that have been conventionally named *hardware layer, controls layer, services layer, components layer,* and *applications layer*.

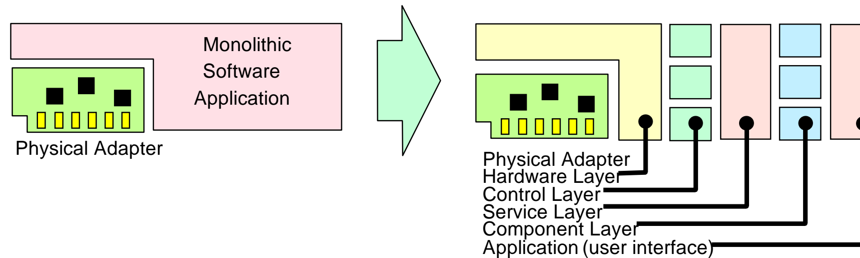

Figure 7. Decomposition of an Application into Independent Parts

The first framework proposal can be summarized as shown in Figure 7.

It became apparent during the analysis of past experience on several different products evidenced that a great number of change requirements can be addressed to parts that could be logically placed in the Service and Component layers without involving changes in the Control or Hardware layers, which are usually the most complex and sensitive.

The Hardware Layer

An asset in the hardware layer is a (C++) class describing a specific low-level interface. It represents a class interface to the adapter APIs. This ensures a useful standardization of the common adapter's features. This level defines a stable set of functions and allows developers to easily implement an adapter upgrade, maintaining compatibility with the existing solutions. A developer should always start here to write software for a new adapter. In practice, this class guarantees a stable abstraction of the low level functionality even when the hardware changes. All of the assets in the hardware layer are C++ classes that inherit from a unique parent class, named **Adapter**. The adapter base class defines a minimum set of characteristics that have been found to be common between all of the physical adapters.

The Controls Layer

This layer consists of the complete set of classes that are involved in the control of the application's functionality at an intermediate level. Assets in this layer should be implemented in a way as much as possible independent from operating system specific functionality in order to make them widely portable. The recommendation is to use ANSI C++ standard coding.

The Services Layer

The typical services asset is a class that represents an high-level interface to a specific device or functionality. This is something like a high level library (API) to use all of the services that are available on top of a hardware adapter, or more generally the services

that can be defined over the functionality of a specific application domain. The designer can start from here to create a user interface or to use a subset of the available functionality. Typically this layer contains more than 90% of the code needed to build a DLL, or a COM/SOM/ActiveX control. To get a working application starting from assets in the services layer requires some coding. The asset that belongs to this layer is a sort of dark gray box. The implementation effort is related to provide a specific interface (stub) over a set of already implemented functionality, retrieved from the *controls* layer.

The Components Layer

This is either a DLL, COM/SOM component, an ActiveX control, or a Java Bean. These components can be used as "black box," without any recompilation, and even used inside a visual development environment, like IBM Visual Age, Microsoft Visual Basic, and others. This supplies a stable, documented, and certified base to build a complete application. To verify the framework, several different components have been implemented over the Datavideo Services class. A dynamic link library, a COM server, and a Java Bean have been implemented easily and in a very limited time frame, the average implementation time being about 2.5 days.

The Applications Layer

An application is seen as the interface between the application domain and the user. The reuse oriented approach tends to identify an application as a "layer" responsible for user interaction. This layer must not interact directly with hardware or a low level portion of code. An application should be made out of high level "bricks"—the components—in order to allow the designer to produce easily a more modular and robust implementation of the required functionality.

Experiment Results

Metrics defined in the early stage have been evaluated. Reported results are not complete, because at the time of writing the project was still ongoing. The available results are sufficiently significant and give a good approximation of the expected results for the whole baseline project. The results have been consolidated for the data transfer protocol that is the application's functional core and covers approximately the 80% of the total reuse activities. The diagrams in Figure 8 show the process metrics results with respect to the Datavideo protocol in its original version.

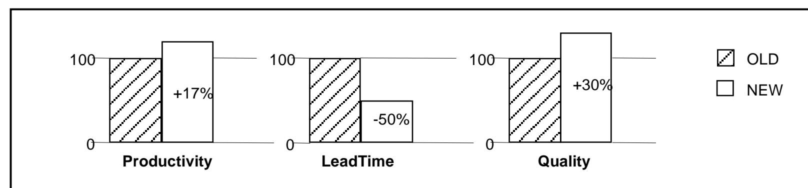

Figure 8. Process Metrics Results

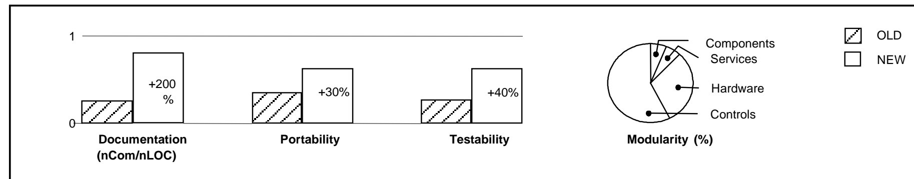

Figure 9. Product Metrics Results

Productivity increased by 17% against an expected value of 25%. The baseline project, however, suffered from the overhead caused by the definition of the framework and its successive refinements. In a long term perspective, this value is expected to further improve. Lead time performed well, giving a 50% reduction against a planned 30%. This result has been achieved both with a real reduction in the development time and with a more effective resource allocation. The quality factor increased by 30%. This result has to be considered as a partial one, because at the time of writing the product had not yet been released to the market. The values shown are referred only to errors detected at design time. It is important to notice that the layered architecture allowed an easier fix of the problems, being they were generally confined inside a single layer.

The use of product metrics allowed a better control over the software complexity. Some of the product metrics, normalized in the range 0 to 1, are reported in Figure 9.

Metrics results on modularity have been shown with a pie chart to put in evidence the decomposition originated by the layered redesign. Most of the functionality goes under the hardware and controls layers, while the services and component layers have mainly interface responsibilities. A better and more comprehensive set of product metrics is expected to be defined in the future. In particular, a considerable effort will be needed to improve automatic data collection and to fine tune the conversion factors.

The segmentation introduced with the layered architecture allowed better identification of responsibilities of the people involved in the design activities, maximizing the effectiveness of specific skills. Each layer can be examined as a specific domain, requiring appropriate knowledge to be properly handled. For example, the hardware layer requires low-level coding skills, but does not require any knowledge about user interfaces. A correct definition of user categories inside the framework architecture is also required in order to protect the inner layers of the solution from access by people who do not have the right domain knowledge to handle them. The framework itself (by means of EuroWare, the Surf Repository for reusable assets) has been enabled to control the access to the right category of users.

Figure 10 shows a summary of the layered architecture, with a gray scale representing the fading to gray visibility of the layers, and showing the main content of each layer and the roles involved in the reuse activities: *developers* are “for reuse” designers producing new assets inside a layer, *users* are “with reuse” designers that build their product starting from reusable assets found in the proper layer.

Note that in the layers hierarchy, a *user* in a certain layer acts as a *developer* in the layer immediately above. For example, an application designer (re)uses the assets from the services layer to develop new assets in the components layer. This made it evident that, in the intermediate layers, development generally includes both “with” and “for” reuse activities.

	Layers	Items	Developers	Users
BLACK BOX	**Applications**	Applications	RAD developer System Integrator	Customers
	Components	DLL library COM/SOM object Active X JAVA Bean RAD object	Application Designer OO Designer	RAD Developer System Integrator
DARK GRAY BOX	**Services**	XxServices class	OO Designer OO Developer Application Expert	Application Designer OO Designer
MID GRAY BOX	**Controls**	XxClass 1 class XxClass 2 class	Application Expert	OO Designer OO Developer Application Expert
LIGHT GRAY BOX	**Hardware**	Xx Adapter class	Dev. Driver Developer	Application Expert

Figure 10. Fade to Gray: Layers and Roles

The introduction of the layered architecture, along with the definition of the design standards in term of coding style, naming conventions, etc., has been a determining factor in improving the quality of the architecture of newly developed systems. The standardization of the design guidelines enhanced the understandability of the design.

From the cultural point of view, the work done in the Surf project resulted in a general improvement in the quality of communications between developers, and UML has been recognized as a powerful way to document projects and exchange information.

Reuse maturity level has been assessed at the end of the project and the results have been compared with the initial assessment. Some differences have been found between the planned and the achieved results. In particular the Surf project performed better than expected in some technical areas, and less than expected in some organizational areas, such as cost estimation. The results are shown in Figure 11, where the initial level is represented with dark squares, the achieved level with striped squares, and the expected level with a heavy black line.

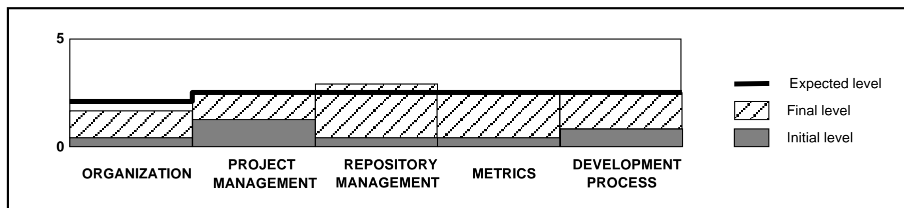

Figure 11. Evolution of the RMM Profile Throughout the PIE: Initial, Target, and Achieved Results

Key Lessons

Several key lessons have been learned in this PIE:

1. *Quality improvements cannot take place without a solid cultural background.* A reuse-based quality improvement needs a sound object oriented mind set, a common language, and a general sense of commitment in order to be effective.
2. *Product metrics evaluation is not a trivial task.* The definition of a significant set of product metrics is not easy. To implement an automatic way to collect metrics data is even more difficult. Again, the definition of the weight of each parameter in the determination of the product quality is a complex task that requires continuous tuning.
3. *Look for benefits in "with reuse" activities.* In the initial phase of a reuse program introduction, the focus goes on the production of reusable assets (design "for reuse"). The organization tends to forget that, in general, reuse generates benefits only in its "with reuse" phase.
4. *A complex repository is more useful in a mature stage.* If the organization is relatively small, and not geographically distributed, a complex repository to store reusable assets is not mandatory in the initial phase of reuse introduction. Because it requires additional efforts for setup and maintenance, it is more suitable for a later introduction. A well structured, shared file system and a version control system can be a good solution in the early stage of reuse practice introduction, particularly if the development team is small and geographically localized.

Conclusions

This PIE faced the challenge of introducing reuse in a software development organization characterized by an heterogeneous environment and without a stable product line. Even if some problems were encountered during the project, the overall result is positive. In particular, the definition of a layered architecture that allows reuse of code at different levels of complexity resulted in a winning choice. Some new projects have been already started following this approach, giving a positive view on future evolution perspectives of this product-based quality strategy.

References

Karlsson, E. A. (ed). *Software Reuse: A holistic Approach*, John Wiley & Sons, Chichester, UK, 1995.

Fowler, M., and Kendall, S. *UML Distilled*, Addison Wesley, Reading, MA, 1997

Garvin, D. "What Does Product Quality Really Mean?" *Sloan Management Review* (26:1), 1984.

WEB Sites Addresses

SURF WEB Page: http://www.ibm.it/surf/index.html
ESSI WEB Page: http://www.cordis.lu/esprit/src/stessi.htm
UML WEB Page: http://www.rational.com/uml/index.shtml

About the Authors

Marco Riva is a product designer in a software development laboratory in IBM Italy. He is responsible for a number of quality improvement projects and for the introduction of new software technologies and development methodologies in the organization. He is also responsible for the PIE described in this paper. E-mail: marco_riva@it.ibm.com

Alessandro Agostoni is a product designer in a software development laboratory in IBM Italy. He is responsible for a number of high technology software development projects and for the introduction of object oriented technologies in the organization. He played a fundamental role in the definition of the layered framework architecture.