Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2007 Proceedings

Americas Conference on Information Systems (AMCIS)

December 2007

An Integrated Framework for Code Reuse and Refactoring in Extreme Programming

Vijayan Sugumaran
Oakland University

Gerald DeHondt
Kent State University

Follow this and additional works at: http://aisel.aisnet.org/amcis2007

Recommended Citation

Sugumaran, Vijayan and DeHondt, Gerald, "An Integrated Framework for Code Reuse and Refactoring in Extreme Programming" (2007). *AMCIS 2007 Proceedings*. 65.
http://aisel.aisnet.org/amcis2007/65

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2007 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

An Integrated Framework for Code Reuse and Refactoring in Extreme Programming

Vijayan Sugumaran
Department of Decision and Information Sciences
Oakland University
sugumara@oakland.edu

Gerald DeHondt
Department of Management and Information Systems
Kent State University
gdehondt@kent.edu

Department of Decision and Information Sciences
Oakland University
dehondt@oakland.edu

Abstract:* *Code reuse has been an area of study in the academic literature for the past three decades and is widely seen as one of the major areas for improving software productivity. Agile development techniques were first developed in the mid-1990s as a code-oriented method of software development that seeks to improve upon the traditional plan-based methodologies. Both approaches bring value to the software development process. The purpose of this paper is to propose a framework that will integrate the strengths of code reuse into the Extreme Programming methodology. It is believed that this approach will lead to a more effective method of software development.

Keywords: Agile Development, Extreme Programming, Software Development Methodologies

Introduction

Agile software development first began in the mid-1990s as an alternative to the traditional Systems Development Life Cycle or plan-based methodologies widely implemented at the time. These lifecycle methodologies take a phased approach to systems development, requiring that one phase be completed prior to beginning the next phase (Hoffer et al. 1998). Agile methods, on the other hand, focus on iterative software development, the continuous implementation of working code. From the beginning of the agile "revolution", specific methodologies have continued to refine these techniques, the most popular being Extreme Programming (XP) first implemented at Chrysler in 1996 (C3 Team, 1998) as a way to accelerate development efforts while producing better software. XP is an implementation of Agile development techniques that is based upon twelve practices, one of which is refactoring. Specifically, refactoring involves modifying software to improve its internal structure in a way that does not alter the external behavior of the code (Fowler, 1999).

Reuse of existing software components has been an area of investigation since the early 1980's and is widely seen as one of the major areas for improving software productivity. By reusing previously tested and implemented code, it is hoped that developers will become more efficient by not having to solve the same problem twice. One of the key challenges with effectively implementing a program of software reuse is the identification of suitable components. If the identification process consumes more resources than saved in development time, these programs will not be undertaken by developers.

Once the appropriate component has been detected, it is then up to the developers to refactor the software into a more suitable solution to the problem at hand. Both techniques, code reuse and refactoring, focus on efficiency in the systems development process. This research proposes an integrative framework that combines code reuse - for component identification - and refactoring - for improved software performance and maintainability - into an overriding process that will improve the efficiency and effectiveness of software development. This framework is also compatible with Extreme Programming, as it improves the refactoring practice or could potentially be used as a method of integrating code reuse as an additional practice within XP.

The remainder of the paper is organized as follows. Section 2 provides a brief review of the related literature, namely, software reuse, refactoring and the need for an integrative approach. Section 3 describes the proposed framework and a potential application in the automotive industry. Section 4 discusses the advantages and limitations of the proposed approach and Section 5 provides the conclusion and future work.

Literature Review

Software Reuse

Software reuse has been an active area of research for nearly three decades. Reuse involves utilizing previously developed software artifacts or knowledge to create new software in new applications. Software reusability is widely seen as one of the major areas for improving software productivity (Sherif and Vinze 1999). A number of reusable artifacts have been developed such as class libraries, components (Szyperski 1998), frameworks, and patterns (Fowler 1997). Research has also been undertaken that attempts to realize the benefits of reuse for object-oriented conceptual design through the creation of tools to facilitate design and construction of new systems with reuse (Sugumaran et al. 2000). Higher-level design fragments and models are being developed (Nord and Tomayko 2006). Clayton et al. (1997) suggest that it would be useful to have a set of previously solved design problems, in the form of domain models that could serve as templates for future designs.

Systematic software reuse provides a promising means to reduce development cycle time, development cost, and improve software quality. Particularly, if reuse is materialized in the early stages of systems development by reusing requirement specifications, designs, and other higher level artifacts, the project team has a tremendous opportunity to drastically cut the overall development time and effort. However, many factors conspire to make systematic software reuse a non-trivial task. For example, contextual and behavioral issues such as management support and software developer's interest are seen to be more important than technical issues in determining the success or failure of a software reuse program (Kim and Stohr 1998).

In order to improve reuse, domain knowledge is essential. Domain modeling captures business domain knowledge at a higher level of abstraction (objectives, processes, actions, etc.), so the artifacts defined at this level can be used to define process requirements (Meekel et al. 1997). Prieto-Diaz (1991) presents a domain model that identifies various domain elements and their relationships, a domain taxonomy, a domain frame (architecture), and a domain language along with standards, templates, interface definitions and a thesaurus. Other research focuses on specific domain level artifacts and associated information for reuse, including processes (specifications, sequence and structure), data (attributes, relationships and structure), and entities (events, agents, and objects) (Chan et al. 1998).

Recently, component based development (CBD) is receiving a lot of attention in systems development. Numerous advantages of CBD are touted. An information system developed from reusable components is argued to be more reliable (Vitharana and Jain 2000), is believed to increase developer productivity (Lim 1994), reduce skills requirement (Kiely 1998), shorten the development life cycle (Due 2000), reduce time-to-market (Lim 1994), increase the quality of the developed system (Sprott 2000), and cut down the development cost (Due 2000). CBD also provides strategic benefits, such as the opportunity to enter new markets, or the flexibility to respond to competitive forces and changing market conditions (Lee 2006). Component providers have the opportunity to enter new markets because of the potential to cross sell components with associated functionalities.

Previous work on CBD has led to the development of repositories for a number of different areas (Sugumaran et al. 2000; Mili et al. 1998). The storage and retrieval of artifacts from reusable repositories is still a challenge, especially when the repositories have many hundreds of components. Several factors impact the search and retrieval process including the scope of the repository, query representation, asset representation, storage structure, navigation scheme, relevance and matching criteria (Mili et al. 1998). Most retrieval methods focus on one or two of these factors to suit the specific domain they are working in. Furthermore, current repositories are limited in that they usually have one representation for each component. When a user searches for a component to satisfy his or her requirements, the user is required to be very specific about what he or she is looking for because some of the repositories do not contain information on how its components are related. Searching for appropriate artifacts does not usually take these relationships, or dependencies, into account and, therefore, limits the usability and consistency of the components returned to the user. Retrieved components are most useful if they respond directly to the user's needs. Although repositories provide artifacts that one could consider at a meta-data level, to be most useful, they need to be augmented with domain specific knowledge because this will provide a means to be able to go beyond a restricted query to identify additional, relevant components for the user. An overall schema of the component reuse process model is shown in Figure 1. As discussed above, the major activities within this process model include domain modeling, meta-data and repository management, conceptual component identification, and component searching and retrieval.

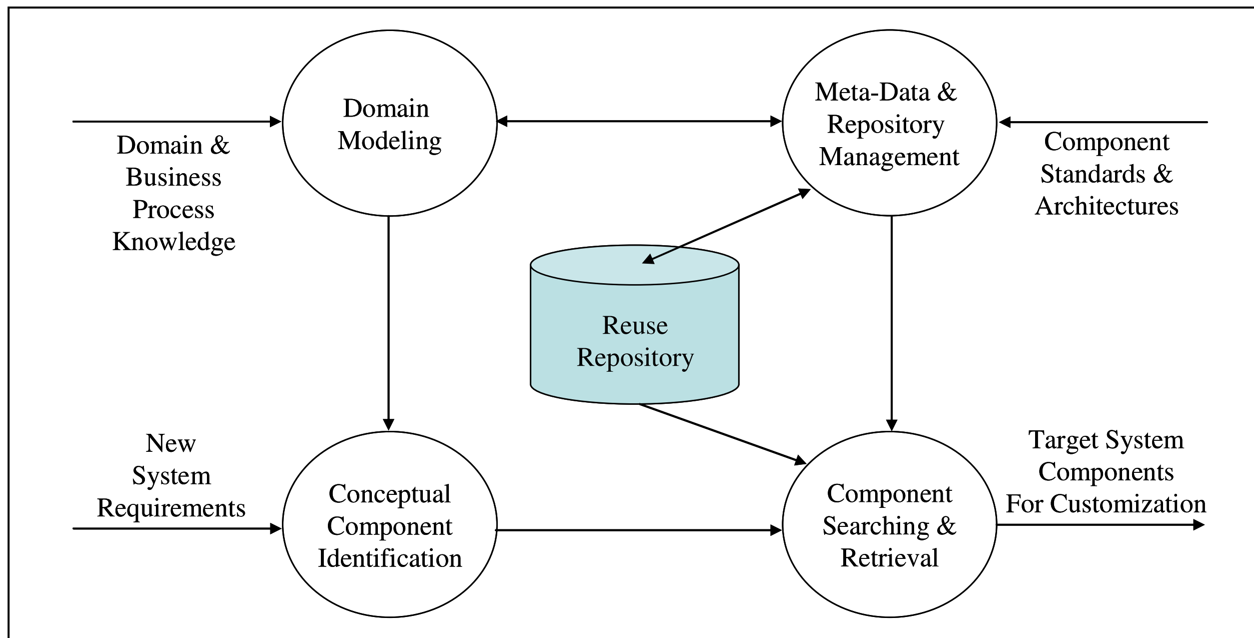

Figure 1. Component Reuse Process Model

Refactoring

One of the most costly areas to maintain in today's corporations is the Information Technology function. A significant portion of this function is the development and maintenance of information systems, with the primary cost factor being the maintenance effort. As software continues along its lifecycle during the maintenance phase, it is modified, enhanced and adapted to new requirements. As this occurs, its complexity increases and it continues to drift further from its original design (Coleman 1994; Guimaraes 1983; Lientz and Swanson 1980). With this in mind, software development efforts and techniques continue to focus on the efficiency and simplicity of code design.

Refactoring, a term first introduced by Opdyke (1992) in his PhD dissertation, has been suggested as a method to help cope with the evolving complexity of these systems. This technique, articulated by Fowler (1999) as "the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure", seeks to restructure the code to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency) (Mens and Tourwe 2004).

Viewed at its lowest level, refactoring is comprised of the following six activities, as articulated by Mens and Tourwe (2004):

1. Identify where the software should be refactored.
2. Determine which refactoring(s) should be applied to the identified places.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
6. Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

Typically, major refactoring efforts are only undertaken once the software has become nearly impossible to maintain (Mens and Tourwe 2004). However, this technique has received significant support from the Extreme Programming community, even going so far as to embed refactoring as one of it twelve principles (Beck 2000). This proactive use of refactoring helps enhance the quality of the delivered software and integrates well with the Test Driven Development (TDD) principle of XP. TDD uses its focus to help keep the code more understandable and maintainable (Nerur et al. 2005). Nerur, et al (2005) also state that TDD facilitates the continuous integration of new code and/or changes without adversely affecting the existing code base. Integrating refactoring into the development approach provides the development team with pre-existing test cases with which to benchmark their efforts.

At its core, XP is fundamentally the continuous delivery of incremental working software that is customer-driven and dependent on communication, incremental design, acceptance of changing requirements, and continuous testing (DeHondt and Brandyberry 2007). The customer-driven focus of refactoring helps integrate it into the XP methodology. Agile methods also emphasize delivering value to the customer early in the development process. This leads to the focus on highly valued features early on in the development cycle (Ramesh et al. 2006). Refactoring of reusable components can lead to shorter development cycles, as reusing components that support valuable features eliminates the effort of developing the component from scratch.

Based on our analysis of the refactoring literature, in particular, the work of Mens and Tourwe (2004), we have developed an overall refactoring process model, which is shown in Figure 2. The driver for this model is the need for quickly generating components by refactoring modules or code segments from prior projects or repositories which might be scattered across the organization. Thus, the main activities within this process model include identifying the sources to refactor, determining which refactoring method to apply, guaranteeing the behavior of resulting components, applying the refactoring method and assessing the refactored components, and storing the components in a reuse repository in a consistent manner for future use.

Overview of XP and the Need for Integration

Extreme Programming (XP) is an Agile development technique that emphasizes frequent feedback to the customers and end users, unit testing, and continuous code reviews. By focusing on rapid iterations of simpler code, XP seeks to identify and resolve potential pitfalls in the development process early, leading to projects that remain focused on the ultimate goal - timely delivery of a well-designed and tested system that meets customer requirements. It has also been described in terms of the values that support it: communication, feedback, simplicity, courage, and respect (Beck and Andres 2004).

XP utilizes an iterative approach that is helpful in developing, modifying, and maintaining systems more quickly and more successfully (Basili and Turner 1975; Boehm 1988). It is these short iterations that provide the flexibility to accommodate the changes requested by the customers and allows the customer to increase competitiveness in the market (Turk et al. 2005). This iterative approach also allows it to be tolerant of changes in requirements (Beck and Andres 2004).

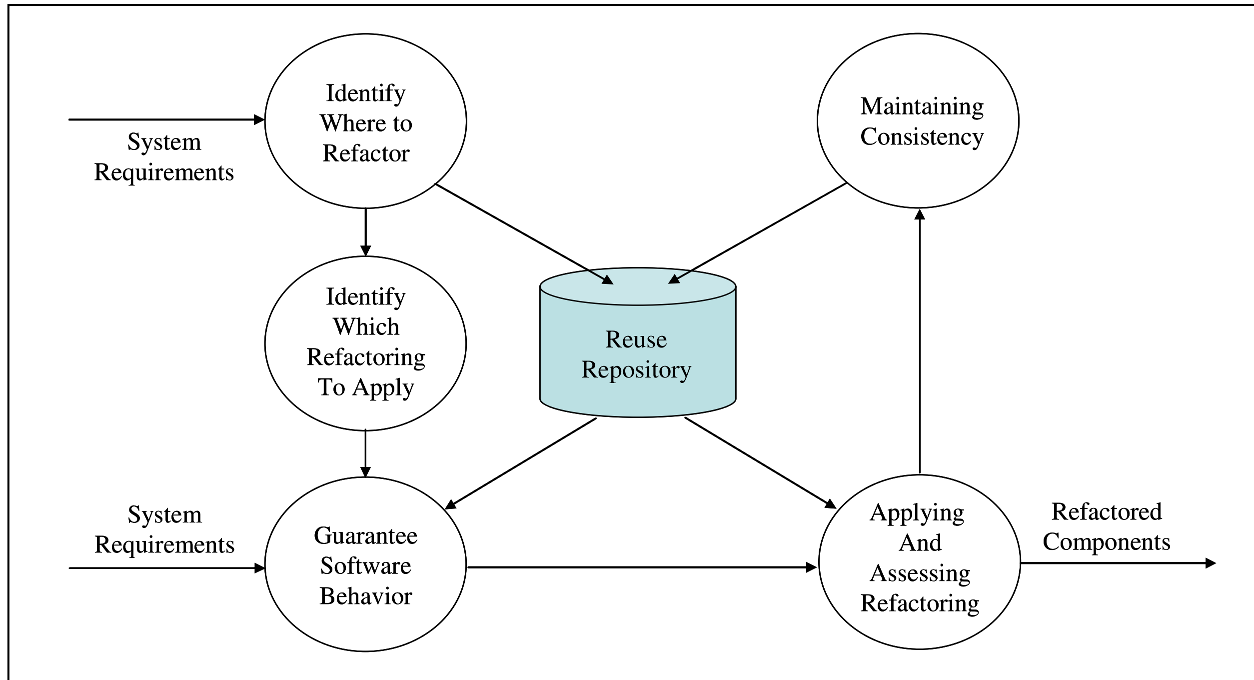

Figure 2. Refactoring Process Model

Inherent practices also include continuous code integration and refactoring to improve the design and code (Nerur and Balijepally 2007). These two ideas form the basis of the integrative framework being proposed. One of the primary drivers of XP is the focus on delivering the features the customer wants. Previously implemented components represent customer requirements that have been sufficiently met. Refactoring and reuse of these components helps continue demonstrated understanding of customer requirements and places developers closer to a completed system. Not only is this explicit communication, but also delivering working code incrementally at frequent intervals. Ultimately, this is the best check to demonstrate understanding of customer requirements.

Integrating code reuse with refactoring increases the efficiency and effectiveness of the systems developers allowing them to continue to meet the needs of the business customer. The following section discusses our proposed framework, which precisely does that. Specifically, it combines the reuse process model and the refactoring process model discussed earlier to create the overall integrated model.

Proposed Framework

While considerable research has been carried out independently in software reuse and agile software development, there has not been much synergy between these two areas. Since the underlying philosophy and goals of each of these fields of research are somewhat similar, namely, developing high quality software in a short duration of time, i.e., minimizing time to market, it is natural that there is cross pollination between these two streams of research. Results from each stream have the potential to significantly impact the other. Thus, in the context of Extreme Programming, developing an integrated approach for code reuse in terms of effectively identifying potential components to reuse and efficiently customizing them to meet the new requirements is crucial.

Domain and business knowledge is critical for systems design and implementation. The encapsulation of business knowledge has several benefits, even if a firm does not move from the analysis to the design phase. First, in today's process reengineering and redesign environments, accessing process knowledge from a repository, especially if it includes best practices, can be valuable for benchmarking and continuous improvement. Second, a firm can capture knowledge about its own business objectives, processes, etc. as part of a repository and continue to revise it, thus providing a source for training new employees. Lastly, if domain knowledge is captured independently of design considerations, design-induced, or representational bias (Buchanan et al. 1983) during requirement analysis will be reduced.

Our proposed integrated process model for reuse and refactoring is shown in Figure 3. It consists of the following steps: a) identifying initial components, b) knowledge-based component searching & retrieving, c) identifying what to refactor &

guaranteeing behavior, d) refactoring & customizing components, and e) consistency checking & managing repository. Each of these steps is briefly described below.

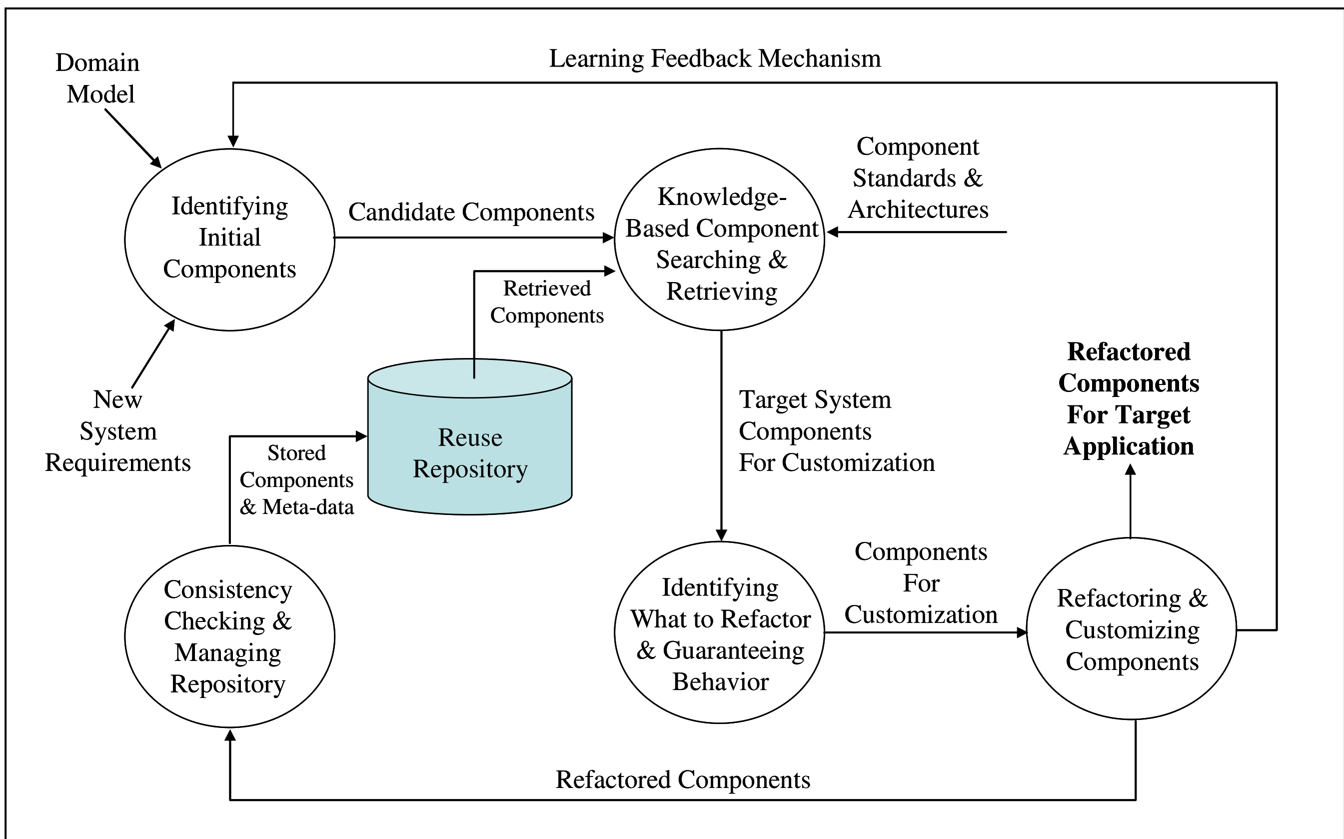

Figure 3. Integrated Framework for Code Reuse and Refactoring in XP

a) Identifying Initial Components: This step involves identifying "conceptual" objects that might be needed in order to support the functionalities specified for a new system. In domain modeling, the typical functionalities that are supported by a system in a particular application domain are modeled as features and appropriate "conceptual" component architectures are developed that specify the appropriate methods and properties for these components. Based on the requirements for the new system, one could begin to identify the types of components that are needed for the target system. Thus, in conjunction with a domain model, the developer can map out the types of components that are required for the target system. This approach is advantageous because the developer does not have to design classes from scratch, particularly, for those classes that already exist. The output of this step is the list of potential candidate classes or components that should be part of the target system to support the desired functionalities.

b) Knowledge-Based Component Searching & Retrieving: This step focuses on retrieving the relevant components from the component repository, based on domain knowledge, component standards and architecture knowledge, and the requirements for the new system. The objective of this step is to move from the conceptual set of components identified in the previous step to a specific set of concrete components that implement the procedures/methods that support the functionalities desired in the target system. To search for components that are the most relevant, the operations of a component are compared to the desired functionalities expressed in the user's requirements. To assess how useful a component would be, we use the closeness measure proposed by Girardi et al (1995). The closeness measure is the similarity value between requirement "q" and a software component "c", which is computed by comparing the frame structure associated with the representation of the requirement and of the software component operations.

c) Identifying what to Refactor & Guaranteeing Behavior: The storage and retrieval of artifacts from reusable repositories is a challenge, especially when the repositories are disparate and contain hundreds of components. Hence, the components retrieved in the previous step may not all be directly usable in the new system. Specifically, the components retrieved may only partially support the requirements. In this step, we focus on identifying which parts of the component are directly relevant and which parts need to be eliminated. Also, the parts that are useful, may not be highly structured or efficient. They may also contain duplicate code. Thus, these components are flagged as potential candidates for refactoring. In addition, to

make these components more effective, their cohesiveness is also examined. If the components are not highly cohesive, they are also potential targets for refactoring. Another important aspect that needs to be enforced during refactoring is the fact that the behavior of the components should not be changed. This step also ensures that the pre-conditions, post-conditions, invariants and temporal constraints are preserved. In other words, the potential refactoring that would be applied to the component should preserve the semantics of the component.

d) Refactoring & Customizing Components: This step focuses on refactoring and customizing the components identified in the previous step by examining the sections of the components that have been flagged for potential changes. Several refactoring techniques exist that use invariants and constraints, graph transformation, program slicing, formal concept analysis, program refinement, software visualization, and dynamic program analysis etc. (Mens and Tourwe 2004). One or more of these techniques can be applied to perform refactoring and generate components that are efficient and meet the requirements for the target system. Components that partially support a requirement may have to be extended or customized to meet the actual requirement. This could involve adding new properties or customizing an existing property. For example, methods within a component may be modified or new methods can be added to extend its functionality. The important point to note is that components may be refactored and customized to fully meet the criteria specified for this implementation, or refactored to more efficiently implement functionality for this scenario. In the former case, the functionality of the component will be efficiently brought in line with current needs. In the latter instance, developers may seek to improve the code while maintaining current functionality.

Traditionally, refactoring and customization are seen as cognitively intense activities. Thus, including them as part of the reuse process model might appear to increase the burden on the developers and prevent them from reusing and refactoring. However, we contend that the available business knowledge and the organizational learning and communication that take place within the organization would alleviate this problem to a large extent. In other words, having a formal learning mechanism and developers actively sharing their experiences and providing feedback would facilitate effective refactoring and reuse. This is an important aspect of the proposed framework. There is a feedback mechanism between this step and the initial step of component identification. As the refactoring step gets executed over a period of time, one can detect similarities and patterns in what and how components get customized and this knowledge could be used in the initial component identification. This learning component will help in better component design and identification. For example, methods unique to the higher level objects (which may not have been identified in the analysis phase) can be presented to the developer for review and possible inclusion.

e) Consistency Checking & Managing Repository: The output of the previous step is a set of refactored and customized components that can be included in the target system. These components satisfy the requirements for the new system to a large extent. However, there may be a need for designing new components from scratch in order to satisfy the requirements that have not been previously encountered. All the components created at the end of the previous step can be stored in the repository for future use. However, we need to explicitly capture and represent the relationships between these components so that they can be better utilized in the future. Thus, consistency checking and repository management is an essential aspect of this approach. When a user searches for a component to satisfy his or her requirements, the searching process should take into account the relationships, or dependencies that exist between components. Or else, it will limit the usability and consistency of the components returned to the user. The retrieved components are most useful for the user if they directly meet the requirements and are consistent with each other.

Potential Application

The proposed approach can be used in a number of application domains where there is considerable pressure to develop new components based on existing components in a relatively short period of time. Many organizations have evolved software development methodologies over a period of time that is specific to their environment and domain. Recently, the software product line (SPL) approach has received a lot of attention. SPL focuses on developing new software components from existing core assets (Sugumaran et al. 2006). For example, organizations such as Nokia, Phillips, Avaya, and GM have reported success in applying the SPL approach to their embedded control software development. One of the major issues faced in SPL is the identification of the most relevant existing components and then refactoring and customizing them to meet the new requirements. Our approach can be very useful in this context. Below, we describe a potential application of our approach in the automotive industry.

Over the years, software has become an integral part of automobiles. Modern cars often embody more than 50 electronic control units with several thousand lines of code running on them. More than 80 percent of automotive innovations are driven by electronics, and amongst them, 90 percent are supported by means of software. One of the key components within an automobile is the powertrain control system, which controls the engine and transmission. The engine control software coordinates fuel, spark and other subsystems to provide greater fuel economy. The transmission control software monitors engine and car speed as well as the loads on throttle, brake pedal, and engine to change gear ratios for an easier and smoother driving experience. The control system architecture comprises different parts related to base hardware, sensors and actuators,

and other electronic devices. The controller software provides the interface between all these devices as well as the algorithms to control and coordinate the timing between them. There are a large number of these components (hardware and software) that have been developed over the years, which are stored in a huge reuse repository. For example, GM Powertrain maintains an extensive repository of design and code artifacts that it uses in designing new control systems.

As new engine types and different transmission requirements get introduced, they require appropriate embedded controls. Whenever a new feature is introduced into an automobile, software engineers have to design appropriate control software to support this feature. This is accomplished by extending existing software components as opposed to developing the component from scratch. In this context, the developer is faced with the task of selecting the most appropriate component from an enormous repository to support the specific requirement. Then, this component is customized and extended by adding additional algorithms or modifying the existing algorithm to meet the timing requirements, etc. The entire control system is tested using simulation. Once the controller meets the statement of requirements, it is then deployed to production control. In this scenario, our approach will be of great help to the software engineer in identifying appropriate components and refactoring them to meet new requirements. The new components developed are then stored in the repository for future use.

Discussion

Improved Software Quality

One anticipated advantage of the proposed framework is the continuous improvement, or organizational learning that is a by-product of this approach to software development. Current refactoring efforts seek to improve the code developed for *the current project*. Code reuse seeks to match software that met user requirements *from previous projects* with similar user requirements from *the current project*. In this way, it is hoped that development efforts are more efficient in not having to re-develop an existing code base. The refactored components also become a part of the organization's reuse repository for potential future use. In line with traditional refactored components, these refactored components would be of higher quality, easier to understand, and feature improved maintainability. It is through this iterative process of continuous improvement that the organization is able to improve the quality of their components. This iterative process will also help improve the organization's efforts of "learning to learn" (Morgan 1998) and fits well with the emerging agile philosophy that values an organization's ability to nurture learning (Nerur and Balijepally 2007).

It is also theorized that this iterative refactoring of reusable components will continue to improve the quality of the components in the reuse repository. Providing an improving reuse repository to developers will also lead to improved software development efforts by the firm. The iterative nature of this approach allows the organization to continuously implement higher quality code in a more efficient manner. Refactoring of the components with each new project will allow the software to evolve and improve over time.

By integrating refactoring into the code reuse process, the organization is able to iteratively improve their software repository, and future development projects.

Framework Caveats

From this process of continuous improvement comes the potential issue of version control, or in this instance, feature control. Once a refactored component is placed back into the reuse repository, it may contain more features than the original component. Identification of this second generation, enhanced component may cause issues with future component identification. Future efforts to identify appropriate components may become more difficult as later generations will have additional features not required by the current project. In these instances an earlier generation of the software may be required to meet the needs of the current project.

Version control in this scenario will require not only tracking between versions, but feature and functionality control as the component develops and improves over time. To effectively manage this process, a system will be required that not only manages each individual version of the software, but also the features added during each iterative generation of the component. This would require additional care in the version management process to adequately recognize the features added in each generation of the software.

As with any new idea, there will be an initial resource investment required to implement the process, train the developers, and identify the component identification ontology. Once the decision is made to enhance current organizational processes by integrating the code reuse strategy with a refactoring approach, the organization will begin climbing the learning curve to effectively implement this approach. With all new processes, there will be an adjustment and learning process to be able to

fully utilize the new approach. Organizations will have to display the courage and fortitude to continue with the process even if initial expectations are unmet. These characteristics will be familiar to XP organizations as one of the values of this approach is courage (Beck 2000).

Another potential issue when implementing a new practice or process is organizational acceptance and buy-in. In order for new techniques to be accepted by the organization, those who will work closest with the process need to accept the process. One method used to obtain buy-in is to first educate developers in how this process will help them be more effective, efficient, and perform their job better. The next step is to provide incentives to the team to utilize this approach.

The belief is that developers will utilize a technique they believe will make them more efficient and effective. The proposed process will serve both purposes if utilized properly. A developer will invest time searching for a component if they believe the search process will be efficient (requiring less time searching for a component than developing from scratch) and effective (time invested searching will yield a better component than they could have developed on their own). An effective code reuse and identification process will allow the developer to readily detect components that will meet their criteria. These refactored components will provide more effective code than could potentially have been developed by the programmer. The efficiency aspect of the utilization model is addressed by the code reuse portion of the proposed framework and the effectiveness portion is addressed by the refactoring approach.

Conclusion

Software reuse has been discussed in the literature for the last three decades and it is still an active area of research within the software engineering community. Similarly, several agile software development methodologies have been proposed and practiced by organizations that develop commercial and non-commercial software. While considerable strides have been made in these two areas of research and practice, there seems to be a lack of synergy between these two streams of research. Much of the agile software development literature and methodologies do not explicitly discuss or incorporate research results from the software reuse literature. Similarly, the software reuse based development methodologies do not take into account some of the guiding principles from the agile development approaches. We have presented an integrated framework for code reuse within the context of Extreme Programming, an agile software development methodology. Our approach combines the salient features of component reuse and agile development and provides an integrated process model for refactoring and reuse within XP. We contend that this integrated approach will facilitate more efficient refactoring, thus improving the overall efficiency of XP. Our future work includes: a) further refinement of our overall framework, b) validating the framework through face validation as well as empirical validation by systems analysts and practitioners, c) applying it to a large scale projects and studying its effectiveness, d) investigating the factors that influence the adoption and diffusion of our approach in organizations, and e) generalizing our approach so that it can be used in conjunction with many agile development methodologies other than XP.

References

Basili, V. and Turner, A. “Iterative Enhancement: A Practical Technique for Software Development,” IEEE Transactions on Software Development (1:4), December 1975, pp. 390 – 396.

Beck, K. Extreme Programming Explained, Boston, Massachusetts: Addison-Wesley, 2000.

Beck, K. and Andres, C. Extreme Programming Explained: Embrace Change, Second Edition. Boston, Massachusetts: Addison-Wesley, 2004.

Boehm, B. "A Spiral Model of Software Development and Enhancement," ACM SIGSOFT Software Engineering Notes (11:4), August 1986, pp. 14 – 24.

Buchanan, B.G. et al. Constructing an Expert System, in F. Hayes-Roth, D.A. Waterman, and D.B. Lenat, eds., Building Expert Systems, Reading, MA: Addison-Wesley, 1983.

C3 Team. Chrysler Goes to “Extremes”. *Distributed Computing*, (October 1998), 24 – 28.

Chan, S. and Lammers, T. Reusing Distributed Object Domain Framework, 5th Intl. Conf. on Software Reuse, June 2-5, 1998, Victoria, BC, pp. 216-223.

Clayton, R., Rugaber, S., Taylor, L., and Wills, L. “A Case Study of Domain-Based Program Understanding” in Proceedings of the International Workshop on Program Comprehension, Dearborn, Michigan, May 1997.

Coleman, D.M., Ash, D., Lowther, B., and Oman, P.W. "Using Metrics to Evaluate Software System Maintainability," Computer (27:8), August 1994, pp. 44-49.

DeHondt, G. and Brandyberry, A. "Programming in the eXtreme: Critical Characteristics of Agile Implementations," e-Informatica Software Engineering Journal (1:1), February 2007, pp. 43-58.

Due, R. "The Economics of Component-Based Development," Information Systems Management (17:1), Winter 2000, pp. 92-95.

Fowler, M. Analysis Patterns: Reusable Object Models, Massachusetts, Addison-Wesley, 1997.

Fowler, M. Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999.

Girardi, M.R. and Ibrahim, B. "Using English to Retrieve Software," Journal of Systems and Software (30:3), September 1995, pp. 249-270.

Guimaraes, T. "Managing Application Program Maintenance Expenditure," Communications of the ACM (26:10), October 1983, pp. 739-746.

Hoffer, J., George, J., and Valacich, J. Modern Systems Analysis and Design. Boston, Massachusetts: Addison-Wesley, 1998.

Kiely, D. "The Component Edge," Informationweek, No. 677, April 13, 1998, pp. 1A-6A.

Kim, Y. and Stohr, E.A. "Software Reuse: Survey and Research Directions," Journal of Management Information Systems: JMIS (14:4), Spring 1998, pp. 113-147.

Lee, J. C. "Embracing Agile Development of Usable Software Systems," CHI 2006, April 22 – 27, 2006, Montreal, Quebec, Canada, pp. 1767 – 1770.

Lientz, B.P. and Swanson, E.B. Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, 1980.

Lim, W.C. "Effects of Reuse on Quality, Productivity, and Economics," IEEE Software (11:5), September 1994, pp. 23-30.

Meekel, J., Horton, T., France, R., Mellone, C., and Dalvi, S. "From Domain Models to Architecture Frameworks," Software Engineering Notes (22:3), May 1997, pp. 75-80.

Mens, T. and Tourwe, T. "A Survey of Software Refactoring," IEEE Transactions on Software Engineering (30:2), February 2004, pp. 126-139.

Mili, A., Mili, R., and Mittermeir, R.T., "A Survey of Software Reuse Libraries," Annals of Software Engineering (5), January 1998, pp. 349-414.

Morgan, G. Images of Organization, San Francisco, California: Berrett-Koehler Publishers, Inc., 1998.

Nerur, S., Mahapatra, R. and Mangalaraj, G. "Challenges of Migrating to Agile Methodologies," Communications of the ACM (48:5), May 2005, pp. 73-78.

Nerur, S., and Balijepally, V. "Theoretical Reflections on Agile Development Methodologies," Communications of the ACM (50:3), March 2007, pp. 79 – 83.

Nord, R. L., and Tomayko, J. E. "Software Architecture-Centric Methods and Agile Development," IEEE Software (23:2), March/April 2006, pp. 47 – 53.

Opdyke, W.F. "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD Thesis, Univ. of Illinois at Urbana-Champaign, 1992.

Prieto-Diaz, R. Domain Analysis for Reusability. In Domain Analysis and Software Systems Modeling, Prieto-Diaz, P., Arango, G. (eds), IEEE Computer Society Press, 1991, pp. 63-69.

Ramesh, B., Cao, L., Mohan, K. and Xu, P. "Can Distributed Software Development Be Agile ?," Communications of the ACM (49:10), October 2006, pp. 41 – 46.

Sherif, K. and Vinze, A. A Qualitative Model for Barriers to Software Reuse Adoption. In P. De and J.I. DeGross (Eds.), Proceedings of 20th Internal Conference on Information Systems, Charlotte, North Carolina, December 13-15, 1999.

Sprott, D. "Componentizing the Enterprise Application Packages," Communications of the ACM (43:4), April 2000, pp. 63-69.

Sugumaran, V., Tanniru, M., and Storey, V.C., "A Domain Model for Supporting Reuse in Systems Analysis," Communications of the ACM (43:11es), Nov. 2000, pp. 312 - 322.

Sugumaran, V., Park, S., and Kang, K., "Software Product Line Engineering," Communications of the ACM (49:12), December 2006, pp. 28 - 32.

Szyperski, C. Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

Turk, D., France, R., and Rumpe, B. "Assumptions Underlying Agile Software Development Processes," Journal of Database Management (16:4), October - December 2005, pp. 62 – 87.

Vitharana, P. and Jain, H. "Research Issues in Testing Business Components," Information & Management (37:6), September 2000, pp. 297-309.