

Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution

Lei Ding

Computer Science Department
University of Southern California
Los Angeles, CA 90098-0781 USA
leiding@usc.edu

Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90098-0781 USA
nenom@usc.edu

ABSTRACT

During the past decade, object-orientation (OO) has become the dominant software development methodology, accompanied by a number of modeling notations, programming languages, and development environments. OO applications of today are increasingly complex and user-driven. They are also developed more rapidly and evolved more frequently than was the case with software systems of the past. All of these factors contribute to a plethora of potential problems when maintaining and evolving an OO application. These problems are caused by architectural erosion, where the initial architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold. We propose an approach, called *Focus*, whose goal is to enable effective evolution of such an application with minimal effort, by recovering its architecture and using it as the basis of evolution. *Focus* allows engineers to direct their primary attention to the part of the system that is directly impacted by the desired change; subsequent changes will incrementally uncover additional parts of the system's architecture. We have applied *Focus* to four off-the-shelf applications to date. We discuss its key strengths and point out several open issues that will frame our future work.

Keywords

Software architecture, recovery, evolution, OO, GUI

1 INTRODUCTION

During the past decade, programming languages and software development environments have undergone dramatic changes. Object-orientation (OO) has become the dominant development methodology, resulting in widely-embraced advances in modeling notations (e.g., UML [2]), programming languages, or OOPs, (e.g., Java [26]), distributed computing technologies (e.g., CORBA [19]), and reusable libraries of software components (e.g.,

Microsoft Foundation Classes, or MFC [15]). The size and complexity of software systems resulting from OO development has grown steadily. Additionally, the predominant computing paradigm has become one in which human-computer interaction plays a key role: applications are increasingly user-driven and characterized by sophisticated graphical user interfaces (GUIs).

These advances have been accompanied by claims of improved developer productivity and software quality, and by an increased frequency and rapidity with which software is evolved ("development and evolution on Internet time"). To minimize their time-to-market, many companies today rely on integrated OO development environments (IDEs) that allow a visual approach to composing and evolving applications from pre-existing building blocks [3,16,17,27]. However, in addition to these implementation-level tools, evolution frequently requires in-depth understanding of an application, its complexity, its overall architecture, its major components, and their interactions and dependencies [1]. Many of these requirements are often ignored in present-day development, with the prevailing attitude that "OO" alone (frequently encompassing nothing more than an OOP and its IDE) gives developers sufficient leverage over an evolution task, regardless of the task's scope and complexity.

This nonchalant attitude, coupled with the complexity of the involved systems, the frequency with which they are changed, and the sloppiness with which the changes are often documented, directly contributes to numerous recorded cases of *architectural erosion* [4,5,9]. Architectural erosion denotes a major departure from the initial architecture's intent and conceptual integrity, caused by its frequent modifications; erosion also refers to the discrepancies between the architecture "as documented" and "as is." Evolving a system with an eroded architecture poses tremendous challenges to engineers and results in a real danger that the modifications intended to provide new functionality will be implemented incorrectly and those intended to remove a particular problem in the existing system will cause other, unforeseen problems.

For these reasons, we propose *Focus*, an approach to *evolving* OO systems that, as a by-product, also *recovers*

their actual *architectures*. The architectures are recovered *incrementally*: only those parts of an application affected by a given change are modified and their architecturally relevant characteristics extensively studied and documented (hence the name “Focus”); the recovery of additional subsystems’ architectures will occur only as new modifications that pertain to those subsystems are required. With each new modification, the task of recovering the architecture of the relevant subsystem and enacting the change becomes easier since a larger portion of the overall system’s architecture is known and correctly documented.

The Focus approach is predicated upon the recognition that, while *implementing* a system using an OOPL may be the best solution, OO is not necessarily the ideal *style* for understanding, capturing, and modifying the system’s *architecture* [20,24]. Instead, in Focus, the architecture is captured using the style deemed most appropriate to the characteristics of the application and its domain. Thus, for example, even though a distributed application is implemented in Java or C++, the architectural style of that application may be client-server [24]. In other words, Focus differentiates between a system’s *logical* and *physical* architectures [14].¹ This aspect of Focus in particular differentiates it from a number of related approaches that only focus on a system’s physical architecture directly recovered from its source code [13,21,23,31].

The Focus approach is based on the assumption that little or no *correct* documentation exists for the system being modified. Additionally, we make a minimal set of requirements:

- 1) The details of the desired modification are known.
- 2) *User-level* properties of the application are known.
- 3) The basic architectural characteristics of the underlying implementation platform (or *substrate*) are understood. An example of this requirement is familiarity with the class hierarchy of the AWT graphics toolkit [6] used in a Java application.

We believe these requirements to be reasonable. The first requirement is not specific to Focus, but is relevant to any software modification task. The second requirement can be fulfilled simply by using an application and observing its behavior. The third requirement may present a somewhat greater challenge to a developer who is unfamiliar with the given implementation platform. However, it is necessitated by the reliance of present-day software development on an increasing body of libraries, frameworks, and middleware solutions.

¹ The logical and physical views of an architecture are often also referred to as *conceptual* and *implementation* architectures, respectively.

While we believe Focus to be generally applicable, for practical reasons we have narrowed down its scope in our work to date. As already discussed, we are currently working with applications developed in an OOPL and with a significant GUI aspect. Three of the four case studies conducted thus far have focused on MFC as the underlying implementation platform. Finally, client-server and C2 [28] are selected as the architectural styles used in an application’s architecture recovery. The choice of client-server was guided by the types of modifications performed in the four studies: in each case the end result was a distributed application. The choice of C2 was a consequence of the nature of the selected applications: C2 was originally designed to support precisely GUI-intensive software systems. This work has thus resulted in two added benefits: the case studies serve as a demonstration of effectively combining two styles in a single application, and provide further data points in the on-going evaluation of C2’s effectiveness in supporting GUI applications.

The remainder of the paper is organized as follows. The details of the Focus approach are discussed and illustrated with an example in Section 2. Two additional case studies conducted to date are outlined in Section 3. Section 4 discusses related work. Section 5 presents open issues, while Section 6 concludes the paper.

2 THE FOCUS APPROACH

The Focus approach is driven by evolution requirements and applied iteratively. Each iteration is composed of two interrelated steps: *architecture recovery* and *system evolution*. Using this approach, the architecture of the original system is recovered, evolved to address new requirements, and enriched with detail in an incremental fashion.

To illustrate the details of Focus, we will use an example application. The application, DrawCli, is implemented in Visual C++ and provided as part of the MFC release. DrawCli provides an editor that allows users to manipulate 2-D graphical objects (lines, rectangles, polygons, etc.). The specific evolution requirement we used as the basis for applying Focus to this system is to extend DrawCli into a collaborative application that allows concurrent drawing and editing of graphical objects by multiple, distributed users. The new system must also provide some level of group awareness, such as recording one user’s actions (e.g., movement, object selection) on another user’s screen and providing a “chat” facility to rapidly discuss issues.

The details of the application, recovery of its architecture, and its evolution are discussed below. We use UML [2] as the notation for expressing different activities and artifacts in Focus. For exposition purposes, we discuss the two major steps of Focus separately. However, we should

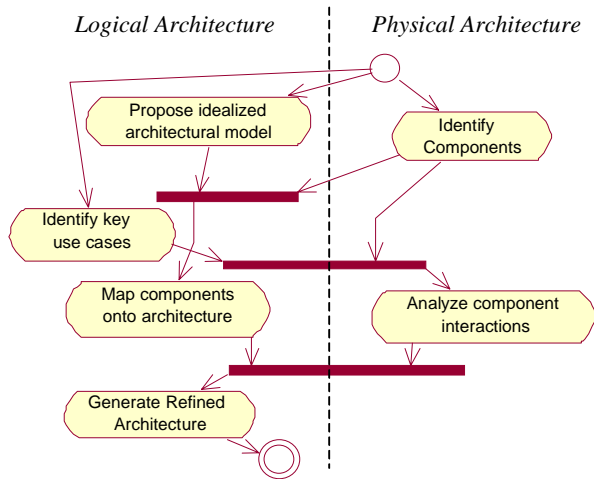


Figure 1. The architecture recovery step of Focus.

reiterate that the two steps are in fact interrelated and applied repeatedly in an iterative fashion.

Architecture Recovery

The architecture recovery step is composed of six activities, as illustrated in Figure 1. These activities are divided into two categories: (1) *logical* and (2) *physical architecture recovery*. The former starts with an *idealized*, high-level model of the architecture inferred from the selected architectural style, the application’s user-level behavior, and its implementation substrate; it focuses on specific parts of the architecture affected by the required change and tries to *refine* them by integrating more concrete details. The latter starts with the source code and tries to

abstract it to get the *actual* components and their interactions in the implementation, in order to inform and influence the architecture refinement activities. The following is a detailed explanation of the activities depicted in Figure 1.

A – Identify Components – A first step in generating the initial logical architecture for a system consists of identifying the coarse grained components from the source code. This step can be further divided into three stages:

- (1) *Generate class diagrams*. No class attributes and methods need to be identified at this stage. However, three kinds of static relationships among classes should be captured in the diagram: association, generalization, and aggregation. A number of tools are available to infer class diagrams from the source code automatically. Figure 2a shows the class diagram of DrawCli, generated automatically by Rational Rose.
- (2) *Group related classes*. The class diagram is further simplified by grouping related classes using several criteria. Classes isolated from the rest of the diagram comprise one grouping (Figure 2b). Classes that are related by generalization (i.e., inheritance) comprise additional groupings, as do classes related by aggregation and composition (Figure 2c). Note that it is possible for multiple application-level classes to inherit from a single base class that belongs to the implementation substrate. In that case, the base class is replicated to facilitate mutually exclusive groupings. Finally, classes with two-way associations are grouped together since they denote tight coupling (Figure 2d).
- (3) *Package classes into components*. Clusters of classes identified in the previous stage are packaged together into abstract components. These components can be further grouped to form even larger components. Using this process, DrawCli’s implementation is abstracted into seven components (Figure 2e).

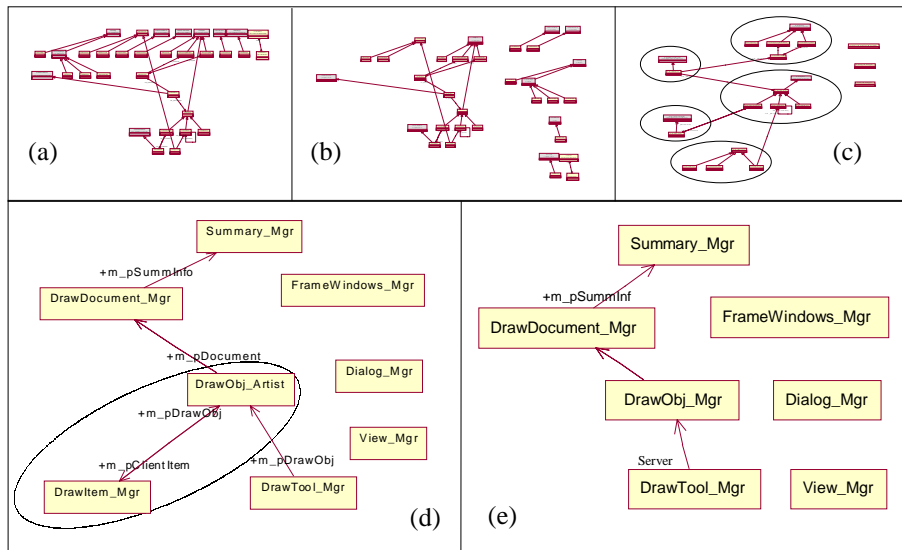


Figure 2. Identifying components from a class diagram. At this magnification, the top three diagrams are shown only for illustration, to convey the scope of the task. Their complexity and the constraint on the paper’s length prevent us from enlarging and discussing them in more detail.

identified in the previous stage are packaged together into abstract components. These components can be further grouped to form even larger components. Using this process, DrawCli’s implementation is abstracted into seven components (Figure 2e).

The above discussion concentrates on the first iteration of the task of identifying components. During subsequent iterations, a subset of these components will be refined to satisfy the evolution requirement. The information captured during this stage is used to ensure that the refinement of the architecture is consistent with the implementation throughout the evolution process.

B – Propose Idealized Architectural Model – During the initial phase of architecture recovery, the

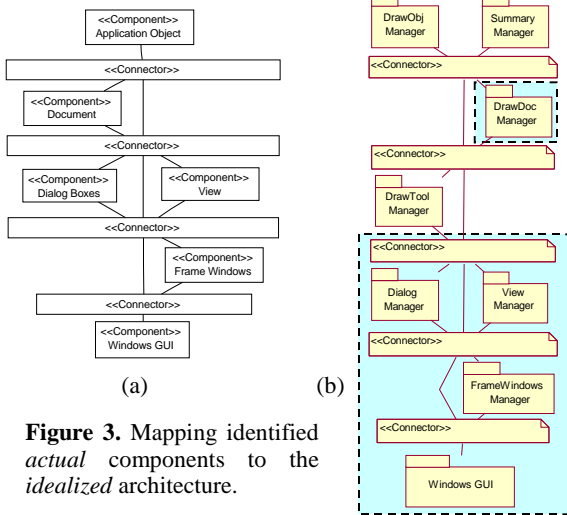


Figure 3. Mapping identified actual components to the idealized architecture.

selection of an appropriate architectural style is critical. Based on that style, the engineer’s understanding of the architectural properties (e.g., program logic, development platform) and user-level properties of the application is captured in an architectural model. This model is *idealized*; parts of it may characterize the application incorrectly. However, the model sets a target for future adjustment and evolution.

Figure 3a represents the initial idealized architectural model for the DrawCli system described in the C2 style [28]. C2 is a style intended for applications with a significant GUI aspect and has been successfully applied in the development of numerous applications to date. In it, an application’s computational elements (*components*) communicate by exchanging messages via explicit services in charge of component interactions (*connectors*). The *topology* of a C2-style architecture constrains the possible interactions in which components may engage. For example, the *Dialog Boxes* and *View* components in Figure 3a cannot directly communicate with each other.

C – Map Identified Components onto Idealized Architecture – Mapping the components identified as part of activity A onto the idealized architecture will yield an intermediate architecture. This architecture only includes those components (e.g., *Dialog Manager*) that clearly fit the idealized architecture, as shown in the two shaded areas of Figure 3b. The remain-

ing components (*Draw Obj Manager*, *Summary Manager*, and *Draw Tool Manager*) will be integrated into the architecture in later steps.

D – Identify Key Use Cases – Focus expresses the major application requirements using UML use case diagrams. Use cases are derived by observing the user-level behavior of the application and from the statement of the desired application modification. When evolving an application, Focus identifies three categories of use cases: those unaffected by the change (e.g., *Print Diagram* in DrawCli); those corresponding to the new requirements (e.g., *Chat*); and those denoting existing functionality that must be modified (e.g., *Open File*). Identifying and prioritizing the latter two categories of use cases sets the *focus* for recovering additional architectural detail and evolving the application.

E – Analyze Components Interactions – To integrate all the identified components into the architecture, in addition to the components’ static relationships (Figure 2), their interactions (i.e., control flow) must be analyzed as well. We use UML sequence diagrams to this end. We rely on two simplifying factors during this activity. First, we describe the control flow at the level of components (Figure 2e) rather than classes (Figure 2a). Secondly, we only generate the sequence diagrams corresponding to the key use cases identified in activity D above. During each iteration of Focus, the remaining use cases with the highest priority are selected. Figure 4 shows an example sequence diagram for drawing a rectangle.

F – Generate Refined Architecture – Once the key component interactions are determined, we can proceed with completing the intermediate architecture created by activity C. We leverage the sequence diagrams to identify the dependencies between the remaining components (from Figure 2e) and those in the intermediate architecture (shaded areas of Figure 3b). We also leverage the sequence diagrams to identify any inconsistencies (e.g., missing interaction links) introduced into the architecture created during activity B. Note that this step will not eliminate all such inconsistencies, but only those relevant to the desired modifications to the application. The resulting architecture is shown in Figure 3b.

System Evolution

Once the application architecture is recovered, we apply the system evolution step of Focus to modify

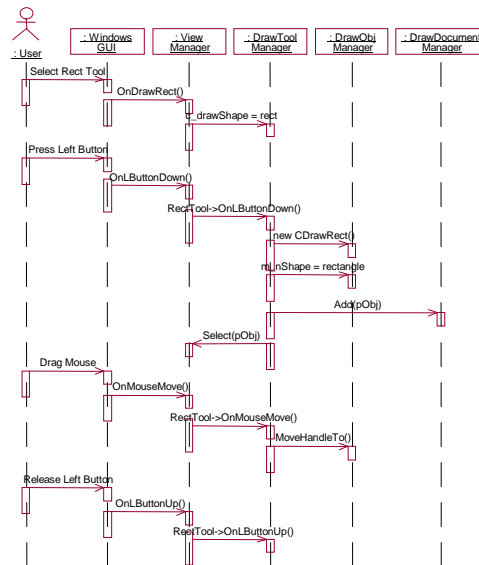


Figure 4. Sequence diagram for drawing a rectangle.

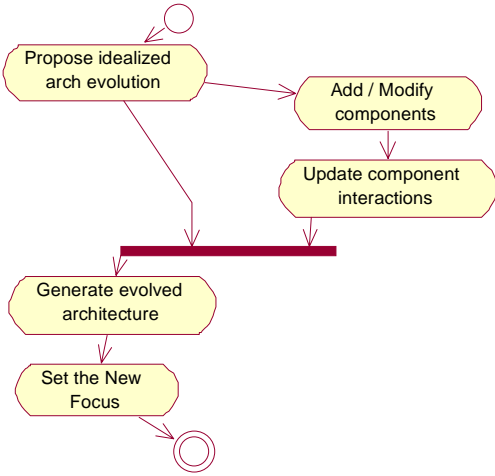


Figure 5. The system evolution step in Focus

the application in a manner that satisfies the new requirements. This step is also carried out in an incremental way, such that the modifications deemed most important are carried out first. As shown in Figure 5, system evolution is composed of the following five major activities.

A – Propose Idealized Architecture Evolution – Before the detailed changes are carried out, a high-level architecture evolution plan is made. In the DrawCli example, we propose to evolve the original system architecture into a distributed client-server architecture, where the internal architectures of both the clients and the server adhere to the principles of the C2 style. We also decide which components from the newly-created clients and server must communicate across the client-server boundaries. In all our case studies, we have found our reliance on explicit software connectors to be very useful in evolving an application.² In this case, we exploit C2’s connectors in order to distribute the application: any connector in the DrawCli architecture (Figure 3b) can be “sliced” into multiple horizontal or vertical sections; each connector section is placed in a different address space and ensures the proper communication flows across the entire connector [7]. Figure 6 shows two such connectors in DrawCli’s evolved architecture, each of which is “sliced” and distributed across the three subsystems.

B – Add/Modify Components – Based on the evolution plan, the engineers must decide whether to add new or modify existing components, or both. For example, in the case of DrawCli, components enabling the chat facility and a server-side component are added to the system as shown in Figure 6; additionally, the interaction aspects of several

² We should note that C2 is by no means the only architectural approach that employs explicit connectors [20,24].

existing components, highlighted in Figure 6, are modified. These modifications are discussed below.

C – Update Component Interactions – As components are added or modified, the control flow among them must be updated. In our example, when one user is drawing a rectangle, to make his actions visible to others, new messages should be generated and sent to the server. The server will then broadcast the messages to other clients. This interaction is modeled via a sequence diagram derived from the one shown in Figure 4; the modified sequence diagram is elided for brevity.

D – Generate Evolved Architecture – All the above changes to the original system should be integrated into the original architecture (Figure 3b). The architecture thus generated will serve as the basis for the detailed design and implementation of the changes. The evolved architecture of DrawCli is shown in Figure 6.

One relevant implementation issue in the DrawCli example is updating component interactions to achieve the distributed drawing capability. This change to the application does not actually require individual components to be modified. Since the new communication among the components must be relayed across the client-server boundaries, we employ connectors to act as interaction intermediaries. DrawCli was originally not implemented using explicit connectors; instead, DrawCli components (i.e., classes) communicate via method calls. Recall that the connectors depicted in Figures 3 and 6 are part of DrawCli’s *idealized*, C2-style architecture. In order to evolve the actual application in the envisioned manner, we introduce explicit connectors in DrawCli’s implementation. Specifically, we have used the technique outlined in [7] to implement connectors based on Windows sockets; in turn, we have used those connectors to achieve the topology suggested by Figure 6.

E – Set the New Focus – If the architecture generated in the preceding activity is still not detailed enough to enable the

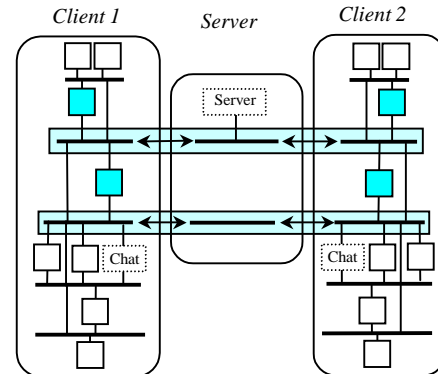


Figure 6. Evolved architecture of DrawCli. With the exception of the added *Chat* component, each client’s architecture is identical to the architecture shown in Figure 3b.

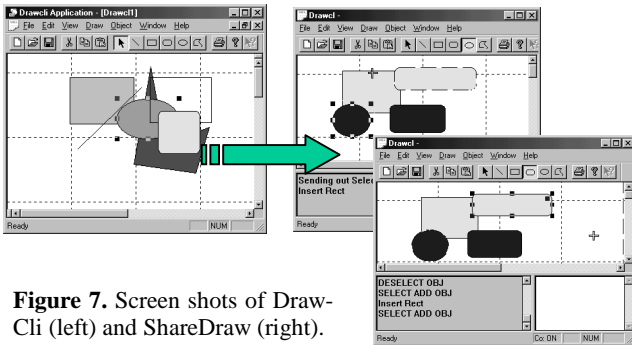


Figure 7. Screen shots of DrawCli (left) and ShareDraw (right).

implementation of the required changes, a new iteration of the two major steps of Focus is required. The components affected by the changes (highlighted components in Figure 6) become the focus of attention during the subsequent iterations; the rest of the architecture will remain unaffected by the changes and is thus ignored in this process. Each subsequent iteration is intended to provide additional, lower-level detail of the impacted components (e.g., their internal structure as indicated in Figure 2) and connectors (e.g., their distribution as indicated in Figure 6). Figure 7 shows the screen shots of both DrawCli and the evolved system, *ShareDraw*.

In summary, the system evolution step of Focus is based on the recovered architecture; at the same time, its results are fed back to (the next iteration of) the architecture recovery step. Therefore, the recovery of the original system’s architecture and the evolution of that architecture are achieved in an incremental, *focused* manner.

3 CASE STUDIES

In this section, we discuss two additional case studies involving architectural recovery and evolution using the Focus method. Our fourth case study entailed evolving the *Microsoft Notepad* application into a distributed, collaborative text-editing tool. In many ways, this case study was conducted as preparation for the WordPad study described below. Although the architectures of Notepad and WordPad proved to be sufficiently different to be of interest to us (e.g., Notepad’s architecture is substantially simpler and the two applications make use of different MFC classes), their differences are ultimately subtle enough to require a more extensive explanation than we feel is warranted. Such an explanation is further complicated by the fact that the two applications can be viewed as providing different variations of essentially the same functionality, so that user-level features cannot be used to easily distinguish between them. For these reasons, we have chosen to omit the discussion of Notepad from this section. The details of the recovery and evolution of its architecture using the Focus method can be found in [25].

WordPad

WordPad is a standard word processing application

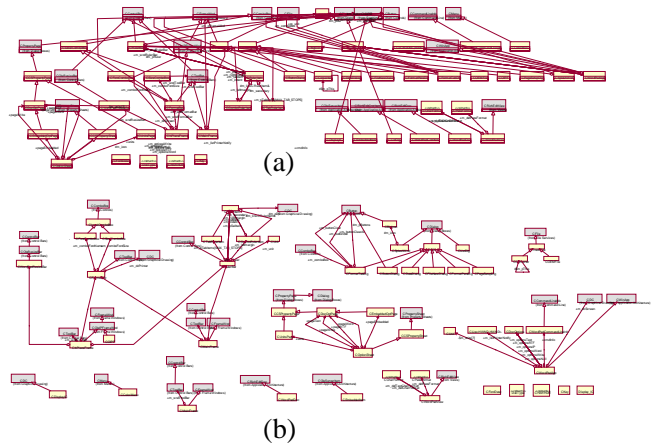


Figure 8. Class diagrams for WordPad: (a) automatically derived from the source code and (b) grouped in the process of using Focus into an initial set of components.

included with *Windows® 98*. It is also provided in the *Visual C++* package as a sample application based on MFC. WordPad supports several “rich format editing” features, including the use of different text styles, fonts, colors, and point sizes; paragraph alignment, tabs, margins, and indentation; and the ability to read, write, and convert among Word 6.0, Rich Text Format (RTF), and ASCII text file formats.

In this case study, the intent was to extend WordPad into a full-fledged collaborative authoring editor that supports fine-grain concurrent editing of shared documents by multiple, distributed users. By fine-grain, we mean that users can even edit the same word concurrently should they so choose.³ Similarly to DrawCli, we decided to integrate the *Chat* component into the evolved application, both as a useful utility in any computer-supported collaborative effort and as a demonstration of the power and flexibility of the Focus approach. Though shown in the evolved architecture of WordPad in Figure 9b (component labeled C), and in the screenshot of the resulting application in Figure 10, this extension is not discussed here for brevity.

WordPad’s key functionality is based on three powerful MFC classes: *CRichEditView*, *CRichEditDoc*, and *CRichEditCtrl*. Compared to a similar system developed from scratch, this rendered the actual size of the application-level code reasonably small: under 20,000 SLOC distributed over 61 source files. However, with over 50 classes and their many dependencies, WordPad’s class diagram is still

³ One may argue against the utility of such fine-grain control over collaborative document editing. However, determining the proper mechanism(s) for accomplishing this task is clearly outside the scope of this work.

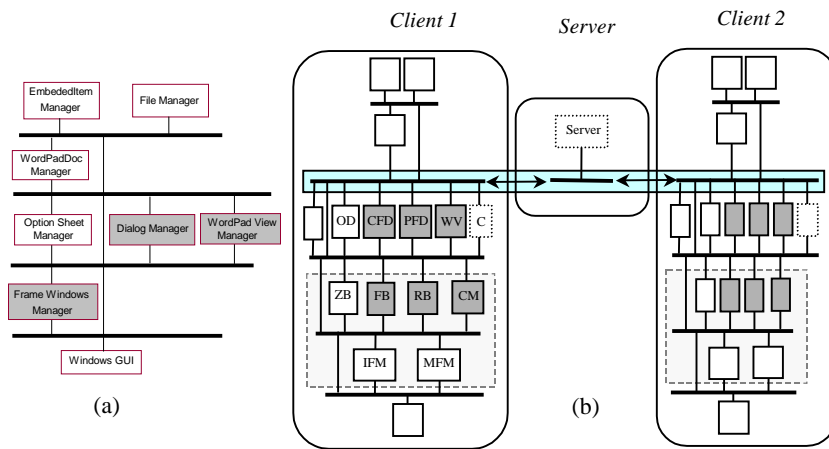


Figure 9. (a) Recovered and (b) evolved architectures of WordPad.

quite complicated (see Figure 8a), and does not provide many hints for modifying the system to satisfy the new requirements.

In applying the Focus approach, the WordPad classes were initially *grouped* into 16 low-level components (shown in Figure 8b). They are then further grouped based on the criteria discussed in Section 2 and, eventually, *packaged* into seven higher-level components. Those components were mapped onto our idealized C2-style architecture for WordPad with the help of several key use cases and their corresponding sequence diagrams. The outcome of this process—the refined WordPad architecture—is shown in Figure 9a.

An analysis of the evolution requirements and the recovered architecture revealed that only the three components highlighted in Figure 9a (*Dialog Manager*, *WordPad View Manager*, and *Frame Windows Manager*) would be affected by the desired change. Additionally, the analysis suggested that, similarly to the DrawCli example, the resulting application be built in the client-server style. For that reason, the *Server* component was introduced into the new system. Again, explicit connectors were exploited to easily facilitate distribution. Figure 9b shows the resulting architecture of the new system after two iterations of the Focus approach.

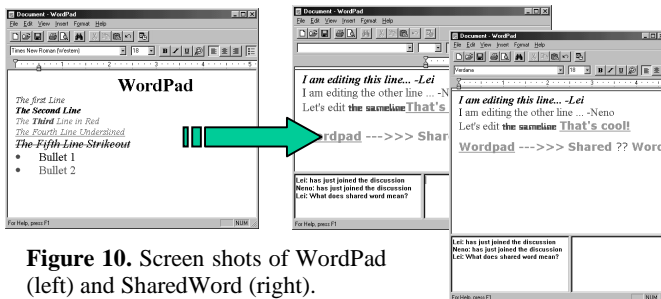


Figure 10. Screen shots of WordPad (left) and SharedWord (right).

The *WordPad View Manager* component from Figure 9a is altered to depict other clients' modifications to a document, and is shown as the shaded WV component in Figure 9b. The *Dialog Manager* component of Figure 9a is refined into three more detailed components in Figure 9b: *Character Format Dialog (CFD)*, *Paragraph Format Dialog (PFD)* and *Other Dialogs (OD)*. This refinement is suggested by the class grouping performed in the architecture recovery step of Focus. Components *CFD* and *PFD* are modified (and thus highlighted in the diagram) to inform other clients of local character and paragraph formatting changes to the document, while *OD* only affects local display and remains unchanged.

Similarly, in each client's subarchitecture in Figure 9b, the shaded area encompassing six components represents the detailed internal architecture of the *Frame Windows Manager* component from Figure 9a: *Resize Bar (ZB)*, *Format Bar (FB)*, *Ruler Bar (RB)*, *Color Menu (CM)*, *In-Place Frame Manager (IFM)* and *Main Frame Manager (MFM)*. Among them, components *ZB*, *IFM*, and *MFM* remain unchanged.

With the help of the evolved architecture shown in Figure 9b, the implementation efforts are focused on the highlighted components and connectors, and leave other parts of the system untouched. Figure 10 shows the screen shots of both the original WordPad and the new collaborative authoring editor, *SharedWord*.

Petri Net Simulator

The first three case studies were based on MFC and, in that sense, it could be argued that they comprise a product family. In order to demonstrate that Focus is more broadly applicable, our fourth case study dealt with a different implementation platform.

In this case study, we modified an existing *Petri net simulation tool*, such that places in the Petri net are depicted by polygons whose number of sides equals the number of tokens inside each place (see Figure 11). Since places with zero, one, or two tokens cannot be represented by polygons, for the purpose of this exercise they are depicted by an empty circle, a filled circle, and a line, respectively. Every time a transition is fired, the shapes of all the places connected to that transition potentially change.

The original Petri net tool was built in Ada using the Chiron-1 GUI management system as its implementation substrate [29]. Analysis of the tool's architecture revealed that it adhered to Chiron-1's strict client-server separation, where the *client* provided the application's logic and GUI,

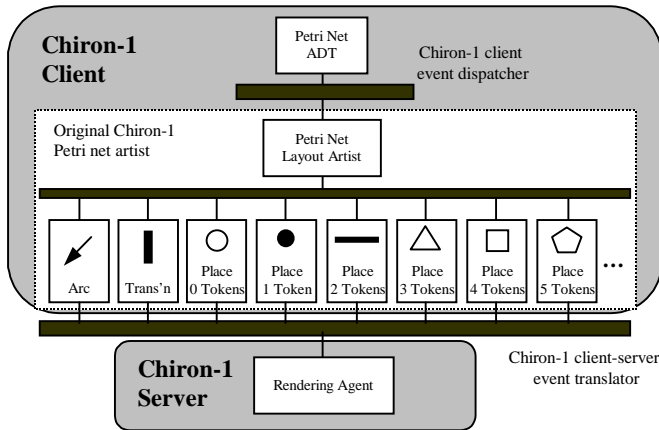


Figure 11. Recovery and evolution of the Petri net tool architecture.

while the *server* was in charge of rendering the application on the screen. Furthermore, the client consisted of two very large components, each of which addressed multiple application needs: an *ADT*, which maintained the state of a Petri net and provided the logic for its execution, and an *artist*, which was tasked with maintaining all aspects of the net's depiction. The ADT and artist communicated via events through Chiron-1's *event dispatcher*.

At this level of abstraction, the architecture of the Petri net tool was easier to recover than was the case with DrawCli or WordPad: the architecture is relatively monolithic and consists of only three coarse-grained components and two event buses (see Figure 11), so that a corresponding class diagram derived by Focus is very simple. However, this presented an added challenge during the evolution step of Focus since any modifications were likely to affect a large portion of the original application.

In order to achieve the specific modification described above and depicted in Figure 11, we redesigned the original artist to fit the C2 style by separating the layout of a Petri net from its presentation. In addition, the presentation of places with different numbers of tokens is entrusted to separate components (i.e., Ada packages). We reused the Chiron-1 event dispatcher to implement the connector between the Petri net layout and presentation layer components. The *Petri Net Layout Artist* shown in Figure 11 maintains the coordinates of places, transitions, and arcs, addresses issues of adjacency, and maintains logical associations with *Petri Net ADT* objects. At the same time, it has no knowledge of the artists in the presentation layer or the actual look of the Petri net. When a place is added, deleted, or repositioned, or its number of tokens changes due to a transition firing, the *Layout Artist* broadcasts the appropriate event notifications and only the artist maintaining the presentation of places with the specified number of tokens responds to them.

In order to achieve this modification, only the highlighted portion of the original application was modified. Most all of the new *Layout Artist's* functionality was reused from the original artist. On the other hand, we had to alter and extend the functionality used for depicting Petri net places in the original application for each presentation-layer artist.

4 RELATED WORK

Many aspects of Focus have been inspired by other work on software evolution and architectural recovery. Like Focus, Murphy et al.'s software reflexion models [18] treat a system's architecture from two perspectives: the idealized, high-level view and the actual, low-level view derived from source code. Moreover, reflexion models allow an incremental approach to architectural recovery: an engineer may analyze whether a desired, but not necessarily complete set of relationships holds between the idealized and actual architectures; the engineer may then repeat the process using a different set of relationships to gain a better understanding of the system. The key difference between this approach and Focus is that Focus is evolution-driven and also provides mechanisms for implementing new requirements. Although reflexion models have been used to aid system reengineering efforts, that is not their primary intended use. Furthermore, unlike reflexion models, Focus makes extensive use of explicit architectural styles and software connectors.

With respect to system evolution, COREM [10,11] is a systematic approach to converting procedural software into OO systems via architectural transformations. It consists of four main steps: design recovery, application modeling, object mapping, and source-code adaptation. Though the steps used by Focus and COREM are similar, their objectives differ: COREM's objective is to change an entire legacy system into the more maintainable OO form, rather than try to incrementally satisfy new requirements.

Similarly to Focus, MORALE [22,23] is used to adapt existing software systems to conform to new requirements. However, unlike Focus, MORALE mainly facilitates the evolution of legacy software systems developed with procedural languages. Furthermore, MORALE is not intended for incremental recovery and evolution.

The details of our three MFC-based case studies (DrawCli, Notepad, and WordPad) closely match the objectives of RENAISSANCE [21]: to support application evolution from centralized to distributed client-server architectures; to support the recovery of system family designs using existing CASE tools; and to support the reuse of sub-systems recycled from existing systems (e.g., the *Chat* facility used in our case studies). The main objective of RENAISSANCE is to prolong the lifespan of large legacy business applications by systematically transforming them into more flexible, evolvable systems. As a result,

RENAISSANCE is a very heavy-weight approach; the description of the approach alone exceeds 300 pages. Another difference between RENAISSANCE and Focus is that the recovery effort in RENAISSANCE does not make explicit use of architectural styles suitable for the chosen application domain.

Recently, a series of studies has been undertaken by Holt et al. to recover the architectures of several large-scale, open-source applications (e.g., Linux, Apache) [4,12]. Their approach is similar to ours in that they also come up with a conceptual architecture and use it as the basis of understanding a system's implementation. However, unlike Focus, their approach makes the assumption that at least some documentation will exist for the system. Furthermore, since the main objective of their work has been system understanding, the reverse architecting process is done thoroughly, with equal effort devoted to each part of a system. In Focus, the recovery effort is centered on the subsystem(s) believed to be affected by the desired evolution. Finally, Holt et al. extract relationships among subsystems (components in our case) via static function call analysis, whereas in Focus this is also done via requirement-driven use-case analysis.

5 OPEN ISSUES

The Focus method has proven successful in our studies conducted to date, showing potential for broader applicability. At the same time, a number of relevant issues remain unexplored and warrant further study. We discuss several such issues in this section.

Architectural Style – In the case studies described in this paper, we have demonstrated the simultaneous use of two styles in an application (client-server and C2). As originally speculated in [28], the chosen styles have proven to be complementary in many ways. Furthermore, we have selected applications for which there exists a well-suited overall style (in our case, C2). Neither of these assumptions will always hold. We need to conduct additional studies before we can assess the ability of Focus to address both (1) additional, less similar styles co-existing in a single application, and (2) multiple candidate styles for an application, none of which is an ideal fit. Further work is also needed to determine the extent to which Focus is applicable to non-OO applications.

Implementation Platform – Three of the four case studies dealt with applications implemented using MFC, thus sharing certain architectural traits. Additionally, one of the authors has extensive experience with MFC, further easing our architecture recovery efforts. The fourth case study (Petri new simulator) has a very different implementation substrate (Chiron-1). While the overall approach still proved effective, our initial lack of familiarity with Chiron-1 required us to study it extensively before we could

proceed with rearchitecting the application. The difference between the two experiences indicates that the implementation platform and the engineers' familiarity with it impact the time and effort required to apply the Focus method successfully. The true extent of that impact is currently unclear.

Application Modifications – Each case study conducted to date centered on a single, well defined modification to a system. If multiple such modifications are needed, Focus currently (implicitly) assumes that the modifications will be carried out in succession, such that the actual architecture of the application is recovered incrementally. However, it is unclear whether a sequential approach is the optimal way of accomplishing such a task. For example, it is likely that, once the subsystems affected by a change are identified, mutually exclusive changes may be pursued in parallel and their results (the recovered architectures and the modified implementations) merged after the fact. Deciding on what constitutes mutually exclusive modifications, merging multiple architectures and implementations that result in the process, and understanding the potential challenges in doing so remain open problems.

Scalability – All our examples to date have dealt with applications under 20,000 SLOC, not counting their implementation substrates (e.g., the MFC classes used by WordPad). As a partial consequence of the sizes of the involved applications, one or two persons were able to complete each case study in a matter of few weeks. While the encountered sizes are representative of many existing applications, further experiments are needed to assess the ability of the Focus method to scale up to larger systems.

6 CONCLUSION

This paper discussed Focus, an approach to recovering and evolving architectures of undocumented, moderately sized OO applications. The approach has proven light-weight and efficient, showing the promise of enabling rapid, yet dependable evolution of a large class of existing applications. The philosophy behind Focus is that architectural recovery is not an end in itself, but is useful only as a means to achieving improved system maintenance and evolution. Focus thus reduces the scope and complexity of the architecture recovery step by focusing the engineers' attention on the system's components critical to the overall task (system evolution). The architecture of any subsystem that has not been impacted by the changes is not "correct," but is rather (deliberately) idealized; the actual architecture of each such subsystem will be recovered only as future changes that impact the subsystem are required.

In addition to its light weight, incrementality, and intentional approximation of a system's actual architecture, another novel aspect of Focus is that it employs the results of software architecture research that have emerged over

the past decade. In particular, unlike other documented architecture recovery approaches (e.g., [13,18,30,31]), our approach directly exploits architectural styles and software connectors. Focus, therefore, not only serves to stem architectural erosion, but by incrementally fitting an application's *physical* (i.e., as-implemented) architecture to its postulated *logical* architecture in the selected style(s), it may fundamentally *rearchitect* the application.

Much work still remains to be done before the true extent of Focus's applicability can be assessed. It is likely that the above-discussed issues of heterogeneous architectural styles and implementation platforms, concurrent application modifications, and scale represent only a subset of the relevant problems. For example, many of the characteristics of Focus suggest that it has the potential to improve upon the state-of-the-art in the architectural recovery of large-scale systems, such as those discussed in [4,12]. At the same time, as demonstrated by Bowman et al. [4], developer discipline is sometimes difficult to ensure in a large system built and maintained by many people over a long time; in such cases, the physical architecture of the system eventually degenerates into a fully connected dependency graph among its components. We have not encountered cases of such severe architectural erosion in our studies to date and it is an open question whether Focus would be able to address them adequately. This suggests to us that both additional case studies and, possibly, further refinements of the Focus approach may be needed before questions such as these can be answered.

REFERENCES

1. K. Araki, ed. *Proceedings of the International Workshop on the Principles of Software Evolution*. Fukuoka City, Japan, July, 1999.
2. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
3. Borland. Delphi. <http://www.borland.com/delphi/>
4. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
5. D. Bredemyer and R. Malan. The Role of the Architect in Software Development. <http://www.bredemeyer.com/pdf%20files/role.pdf>
6. P. Chan and R. Lee. The Java Class Libraries: An Annotated Reference. Addison-Wesley, 1996.
7. E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *21st International Conference on Software Engineering*, May 1999.
8. S. Demeyer and H. Gall. Report: Workshop on Object-Oriented Reengineering (WOOR '97). *Software Engineering Notes*, January 1998.
9. W. Eixelsberger, M. Ogris, H. Gall, and B. Bellay. Software architecture recovery of a program family. In *20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.
10. H. Gall, R. Klosch, and R. Mittermeir. Object-Oriented Re-Architecting. In *5th European Software Engineering Conference*, Berlin, September 1995.
11. H. Gall and J. Weidl. Resolving Uncertainties in Object-oriented Re-Architecting of Procedural Code. Technical Report TUV-1841-98-01, T.U. Vienna, 1998.
12. A. E. Hassan and R. C. Holt. A Reference Architecture for Web Servers. To appear in *Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000.
13. R. Kazman and J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *5th International Conference on Software Reuse*, Canada, June 1998.
14. P.B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, November 1995.
15. Microsoft. Microsoft Foundation Class Library. <http://msdn.microsoft.com/library/devprods/vs6/visualc/vcedit/vcrefwhatsnewmfcvisualc6.0.htm>
16. Microsoft. Visual Basic. <http://msdn.microsoft.com/vbasic/>
17. Microsoft. Visual C++. <http://msdn.microsoft.com/visualc/>
18. G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.
19. Object Management Group. Common Object Request Broker Architecture. <http://www.corba.org>
20. D. E. Perry and A. L. Wolf, Foundations for the study of Software Architecture. Software Engineering Notes, October 1992.
21. The RENAISSANCE Web. <http://gateway.comp.lancs.ac.uk/projects/RenaissanceWeb/>
22. S. Rugaber. An Example of Program Understanding. Technical Report GIT-CC-98-14, Georgia Tech University, December 1997.
23. S. Rugaber. A Tool Suite for Evolving Legacy Software. In *International Conference on Software Maintenance*, Oxford, England, August/September 1999.
24. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
25. Software Architecture Recovery with *Focus*. <http://sunset.usc.edu/~nen0/Focus/>
26. Sun Microsystems. Java. <http://sun.java.com/>
27. Symantec. VisualCafe. <http://www.visualcafe.com/>
28. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
29. R. N. Taylor, K. A. Nies, G. A. Bolcer, C. A. MacFarlane, K. M. Anderson, and G. F. Johnson. Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support. *ACM Transactions on Computer-Human Interaction*, June 1995.
30. V. Tzerpos and R. C. Holt. A Hybrid Process for Recovering Software Architecture. In *CASCON 1996*, Toronto, Canada, November 1996.
31. K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, January 1995.