Abstract

Dual lifecycle software processes have the potential to significantly improve the way in which suites of software applications are generated and sustained. However, several outstanding issues need to be more adequately addressed before the full potential of this philosophy can be realized. Detailed strategies for maintaining domain architectures in parallel with suites of fielded applications are at present particularly conspicuous by their absence. In this paper we present a dual-lifecycle maintenance process that was developed for the ROSE project, a major reengineering and repository-building effort in the domain of Flight Design and Dynamics. We present the major features of the process, the rationale behind these features, and changes which we feel would be beneficial based on lessons learned from the application of the process. The process is presented using a variant of the Fusion object-oriented design method known as ProFusion.

1 – Introduction

Organizations which develop software rarely use or sell just one program. Typically, they deal with a suite of related applications, each meeting certain distinct requirements while at the same time sharing a core set of common properties. In the past, such applications were typically developed and maintained in isolation from one another, leading to significant duplication of effort in all phases of the software lifecycle. Recently, however, a new paradigm has emerged which seeks to exploit the commonality between applications. The “dual lifecycle” approach separates the activities involved in software development and management into two distinct groups within the organization: those concerned with the domain as a whole, and those concerned with particular applications within the domain (product family). The domain and application engineering activities are distinct, yet related, endeavors which proceed in parallel.

The group concerned with domain engineering activities is responsible for the common aspects of the product family, such as analyzing, designing and implementing an overall architecture for the family, and for populating a repository of reusable components. In contrast, the application engineering group adapts (preferably through extension) the domain architecture into finished concrete applications. The adaptation process is a complete development process in itself, but it is also jump-started by the domain products.

Over the past three years, as part of the Repository Based Software Engineering (RBSE) project [10], we have defined a dual lifecycle process (shown in Figure 1) [11] for the Reusable Object Software Engineering (ROSE) project, a major NASA reengineering effort lead by Rockwell Space Operations Company to develop a repository of reusable objects in the domain of Flight Design and Dynamics.

In this paper we describe how the dual lifecycle process developed for the ROSE project addresses maintenance issues. After providing a little background in section 2, we identify the special problems which arise when addressing the issue of maintenance within a dual lifecycle process (as opposed to the basic development phases such as analysis, design or implementation) in section 3. Before discussing the actual maintenance process, we use section 4 to provide a brief introduction to ProFusion [4], a process modeling technique developed in the RBSE project and based on the Fusion development method [7]. We use ProFusion throughout this paper to describe our processes. Section 5 presents the dual lifecycle maintenance process and sections 6 and 7 contain detailed discussions on Domain Mainte-
nance and Application Maintenance, respectively. Finally in section 8 we draw some conclusions and point out some areas for possible future research.

2 – Background

Although the general dual lifecycle concept has been around for some time [20], little research has been published on the details of the individual phases, particularly the maintenance activity which is the most difficult of the traditional lifecycle phases to adapt to this new paradigm. The experience factory concept proposed by Basili, et.al. [5] includes iterative adaptation of software processes and products, but no focus on domain concepts (except for the encompassing notion of an experience base for the organization). Yeh’s work in concurrent engineering [22] is distinctly resonant with ours, but focuses on software process reengineering and improving levels of concurrency within a more traditional perspective of the software lifecycle.

Unlike the earlier development phases in Figure 1, maintenance at the application level has an impact at the domain level. A change requested to a domain component for the needs of a given application could have an impact on all other applications which use that component for the needs of a given application could have an impact on all other applications which use that component. In contrast, analysis, design, and implementation at the application level have no required impact on the work carried out (i.e. the reusable components generated) in the corresponding domain activities. Consequently, the issues that confront us here are not those of instantiation of systems comprised of architectures, components, etc. [1, 16], but rather those that are more akin to the problems that arise when maintaining large object class hierarchies [8, 14, 15, 17] or integration of COTS packages into application systems [23].

Additionally, the ongoing maintenance activities at the domain level must always include careful analysis of downstream effects in fielded applications. There is consequently a much closer relationship between the domain and application maintenance activities than for any of the other activities. Moreover, the long term success of the dual lifecycle approach is critically dependent on the success of these related maintenance activities. Yeh’s perspectives on concurrent engineering [24] hold interesting promise here. The code health model [3] for traditional (i.e., single lifecycle) maintainability is also relevant.

Finally, maintenance activity at the application level must always include a careful analysis of the potential for incorporation into the domain level. This relates strongly to issues of domain model - based reverse engineering [9].

3 – The maintenance challenge

With a dual lifecycle approach to software development, the composition of any fielded application is a mixture of general and specific components created either by the domain or applications engineering group. Each group retains ownership of its products. In particular, domain engineers develop, maintain and enhance their products with a view to satisfying all their customers (applications using domain components).

The basic challenge facing maintainers of applications built using such an approach is that changes made in fielded applications may necessitate corresponding changes in domain products. Unfortunately, changes to domain products may impact more than one application. In other words, maintenance of a single fielded application, when performed poorly, can degenerate into simultaneous maintenance of the application, domain products and other applications that depend on them.

Of the unique new challenges posed by maintenance in this paradigm, we found several which were particularly difficult to address: timing issues, separation of concerns and cost structure issues.

3.1 – Timing issues

First and foremost, maintenance in a dual-lifecycle approach must take into account that there are multiple lifecycles involved, each of which is on its own asynchronous thread of control. Trivial as it may seem, this issue invalidates many common assumptions made when designing maintenance processes. Activities that cross application/domain boundaries are heavily time-penalized. The time penalty manifests itself both in terms of delays in the communication mechanisms employed and in resource allocation difficulties.

For instance, if an application maintenance team makes a change request for a domain product, there is a certain amount of time that is “lost” due to the latency involved in the domain maintenance (requests are queued through change management). The application maintainers also have a long stretch of time during which they are waiting for domain maintenance’s response -- hence calling for a reallocation of their attention to other requests in the interim. This issue obviously also arises within regular maintenance processes in the case of handoffs between areas of specialty (e.g., analysis, design, etc.) but their importance grows with the granularity of the time delay. These timing issues particularly affect the impact analysis and release planning maintenance activities.

3.2 – Separation of concerns

Application and domain maintainers have vastly different concerns, yet must work together towards a common goal - fielded applications. The goals of application maintainers are tightly focussed on a single application while domain maintainers need to juggle costs and benefits across all fielded applications plus foreseeable future ones within the domain. These two outlooks are often contradictory.

Trying to correctly classify a change as truly domain or application specific can be much more complex than it appears. What seems very much like a change to a domain component (from an application maintenance perspective) may in reality be a very application specific extension to a domain component. So, the overall decision-making process must carefully disallow the inherent “selfishness” of application maintenance from undesirable domain-wide impacts while also permitting local changes that run contrary to the overall scheme of
things but nonetheless are of capital importance to individual applications. Furthermore, the decision-making process has to minimize needless interactions, since each cross-life cycle interaction is accompanied by a time penalty (see the timing concerns, above).

3.3 – Cost structure

Given that the application and domain maintainers’ priorities can be at odds when evaluating changes, the maintainers should share a common cost model that allows decisions to be taken with a global view. The use of a shared cost model allows domain maintainers to “import” resource and cost estimates from application maintainers and use them as part of an overall estimate. A secondary issue here is that it’s quite common for fielded applications not to pick up new versions of their domain components every time there is a new release. However, when an application maintenance team requests a change to an older version of a domain component the cost must include the cost of bringing the application up to the latest version, plus the change required.

4 – ProFusion

We describe the ROSE maintenance process using ProFusion [4], an object-oriented approach to process modeling based on the Fusion software development method [7]. In ProFusion, the four commonly recognized facets of a software process (informational, functional, behavioral, and organizational) are described using three tightly-integrated Fusion models (object models, operation schemata and object interaction graphs). The informational and organizational aspects of the process are captured using an object model, and the functional and behavioral aspects are captured using a combination of object integration graphs and operation schemata.

A small number of simple extensions have been defined to enable ProFusion to describe concurrent processes, such as the ROSE domain maintenance process. In particular, independent threads of control are depicted using a special “thread” icon within object-interaction graphs. This takes the form of a circular arrow resembling a “loop.” Depending on its position within an object-interaction graph, a thread icon can either depict an active object which can send messages spontaneously, or an asynchronous interaction which is serviced concurrently with the client. Additional annotations allow timing constraints and exceptions to be described [4].

Two variations of standard Fusion models are used to describe how concurrent invocations of a given object are interleaved. A participation graph shows all the messages which an object sends or receives, and is easily derived from the set of interaction graphs in which the object appears. An object lifecycle model defines the order in which the messages arriving at a given object should be serviced. This model uses the same notation as the general Fusion lifecycle model, but applies only to an individual object.

5 – A dual-lifecycle maintenance process

In this section we present the maintenance process. The overall lifecycle process developed for the ROSE project was highly influenced by the Clear Lake lifecycle model [21]. A well-defined interaction scheme between application and domain maintainers allows the application and domain concerns to remain separated while maintaining the organization’s ability to maintain fielded products. Unlike the simple maintenance case where there is a supplier and a customer, the domain and application engineering teams have additional internal supplier/customer relationships within the organization itself (one for each application group).

5.1 – The maintenance process template

Participants

In a traditional maintenance process the participants include a single team of maintainers whose purpose is to respond to requests of fellow developers and the customer(s). In the case of a dual lifecycle, the number of participants increases while their interactions become more numerous:

- Domain maintainers respond to requests from domain developers, application developers and application maintainers. They may also make requests to the application maintainers when domain product changes have impacts on existing applications.
- Domain engineering developers may find errors in or propose improvements to fielded portions of the domain architecture already under the control of domain maintainers.
- Application maintainers respond to requests from application developers, the customer(s) and domain maintainers.
- Application engineering developers may find errors in or propose changes to the fielded portions of the domain architecture that they are attempting to incorporate into their application. Since applications can be complex, involve a phased delivery and interact with other applications in the domain, application developers are quite likely to make requests to application maintainers within their own group or to other application groups.
- Customers, of course, only have an outside view of the development process and report the errors they find and the improvements they would like to the one contact point they have: the application maintainers.

In other words, the use of a dual lifecycle introduces an extra degree of complexity to the basic client-server relationship vis-a-vis maintainers of software systems.

Process steps

The dual lifecycle maintenance process adopted in the ROSE project, and described in this paper, is adapted from the maintenance model presented by Arthur [2]. This model identifies 6 primary steps in the maintenance process, shown in Table 1.
Since we assume that the reader is very familiar with the general contents of maintenance processes, we shall not elaborate greatly on these steps here. We will also use the traditional naming convention of corrective, adaptive and perfective change requests to describe the three major kinds of requests processed by the maintainers. The traditional maintenance process can be described at a high level with the object interaction graph in Figure 2.

Each change request has its own thread of control and the change management step is performed asynchronously with respect to the individual requests. In this model, both the users of a system and its maintainers may submit change requests to change management. The maintenance team must then perform the impact analysis in order to determine the ramifications to the system and resources required in order to effect the change. This information, developed from the current system and cost model, is then used by the configuration control board (CCB) to make a decision on the change. Although the sequence shown above is intended to be applied in a traditional software lifecycle, it is readily tailored to both domain and application maintenance and developed the interaction points necessary for the synchronization between them.

Figure 2: The traditional maintenance process

Table 1: Arthur’s maintenance process

<table>
<thead>
<tr>
<th>Change Management</th>
<th>M_Team : M team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impact Analysis</td>
<td>(1)</td>
</tr>
<tr>
<td>Release Planning</td>
<td>(2)</td>
</tr>
<tr>
<td>Corrective</td>
<td>(3)</td>
</tr>
<tr>
<td>Adaptive</td>
<td>(4)</td>
</tr>
<tr>
<td>Perfective</td>
<td>(5)</td>
</tr>
<tr>
<td>Code Changes</td>
<td>(6)</td>
</tr>
<tr>
<td>Testing</td>
<td></td>
</tr>
</tbody>
</table>

change management | identify, describe and track requested changes
impact analysis   | determine scope of the change, and its potential impact (positive and negative)
plan system release| determine the content and timing of system releases
design changes    | revise the design for the system
code changes      | change the software as approved
testing           | verify the changes
5.2 – Domain maintenance

The basic steps in the domain maintenance process follow closely from the steps in the traditional maintenance template. The differences reflect the needs associated with evolving a domain architecture which remains synchronized with the requirements of its product families. The domain maintenance process can be described at a high level with a very similar object interaction graph, as shown in Figure 3. The decomposition of several of the activities is quite different, however. During the Change_Management step the domain maintainers must verify (to the extent possible) the request’s domain specificity in addition to its type (corrective, etc.). The next, and most noticeable difference, occurs in the Impact_Analysis step. Tracing request impacts and estimating their necessary resources are potentially substantially more time-consuming than in the traditional maintenance process. Tracing the impacts must not only be done for the domain architecture, but also for each of the fielded applications which could be affected by the change. Resource estimation is further complicated by the fact that there are potentially multiple resource estimates to be performed, that for the domain architecture itself plus those for the applications impacted by the change. We have also added a step entitled Analyze_Changes following the type specific (corrective, etc.) steps. This step’s objective is to make a pass through the Domain Analysis process in order to make sure that all analysis changes implied by the change requests approved for the next release are captured and incorporated into the domain analysis product set. We have also used the three type specific (corrective, etc.) steps to perform predesign activities specific to the type of request. This requires that we have an additional Design_Changes step in which we perform domain design for the change(s). The final difference is the inclusion of testing within the Implement_Changes activity rather than a standalone activity, following the principle that something isn’t done until it works.

5.3 – Application maintenance

Application maintenance varies from the basic maintenance template in a few, localized ways. These differences do, however, have profound influences on the maintainers’ work. The two areas with the most visible changes are those of impact analysis and release planning.

The application impact analysis process step differs from the general maintenance template in the introduction of an extra activity pertaining to the determination of impacts that application changes have on domain products. Specifically, the existence of such impacts needs to be ascertained by the application maintainers and their extent evaluated by the domain maintainers themselves. At the very least, the application maintainers need to pay particular attention to the impacts of application changes to domain products since, unless exceptional actions are taken, there is no way for the application maintainers to make them.
Figure 4 shows the impact analysis activity for one application change request. The most notable top-level variation is the addition of the **Trace_Domain_Impacts** activity. This activity encapsulates the steps taken by application maintainers to determine the impact on domain products of making the application change. This includes determination that there is such an impact (e.g., that part of the domain architecture needs to be reworked in order to support the planned application changes), their expression as domain change requests and the handoff to domain maintainers in order to gauge their feasibility and cost. Having gathered this data, the following steps of the process are basically unchanged from the template.

Assuming that a change to the application has domain impacts, each corresponding domain change request eventually is either approved (with an associated cost) or rejected by the domain maintainers. Given this information, application maintenance can proceed by relying upon the domain changes or seek workarounds/remedies to domain changes that are rejected or that incur too high an implementation cost. This decision is taken by the application maintenance CCB.

Changes to the release planning step are a direct reflection of the increased complexity of scheduling an application maintenance release when portions of the work may be implemented as part of domain maintenance. In particular, updated domain components may only become available as part of a future scheduled release of the domain architecture, thus limiting the time frame when work using them may begin. The release planning process step for application maintenance thus requires a more advanced work scheduling methodology but is not fundamentally changed otherwise.

Figure 5 expands upon the domain impact determination activities shown on figure 4. Note that the corrective maintenance-specific error localization steps that are part of the resolution of changes in the generic maintenance process are moved forward in the application maintenance process in order to allow the maintainers to make an informed decision as to the location of the error. It is of great import when errors to domain products need to be passed on to their respective maintainers.

### 6 – Domain maintenance

Some of the more challenging issues facing the domain maintainers include temporal requirements related to ability to meet the schedules of a given application, classification of proposed changes as truly domain or application specific, and backward compatibility of a change(s) with other fielded applications within the domain.

#### 6.1 – Timing issues

The temporal requirements related to the ability to meet the schedules of a given application must be analyzed during impact analysis (see Figure 6). The **Estimate_Resources** (4) and current domain maintenance delivery schedule must be evaluated to determine if the change could be made in time to be of use to the application maintainers. If not, dispositioning the request becomes difficult. It is possible that changes to domain products may be required in an application maintenance context, while the same changes may be desired at the domain level, but placed in a later release plan.

This scenario poses several challenges resulting from work on a domain product at two potential points. Application maintenance may be forced to implement the change on their own because it is essential, but the
domain maintenance team is unable to complete it by the required date. This opens up the issue of application maintenance modifying a domain component(s), creating a synchronization problem with the domain. The domain maintenance team may also determine that while the change could not be made in time to benefit a particular application, it is nonetheless beneficial to the domain as a whole and schedule it in a future release. This also creates a potential synchronization problem as domain maintenance works a change (in a future release) which may by then have been addressed by one or more fielded applications.

To avoid long term divergence, we decided that any change request which is determined to involve a domain component, but made by an application maintenance team, must result in a subsequent perfective change request to domain maintenance. Additionally, any change request which cannot be completed by domain maintenance by its need by date must be rejected. Rejection by this rule causes the creation of a new domain change request (which references the old one) for consideration in a later release. The impact analysis on this new change request must include assessment of any application solutions, which would have generated their corresponding perfective change requests.

6.2 – Separation of concerns

The issue of classifying proposed changes as truly domain or application specific is first encountered in the Change_Management step. An initial determination must be made at this point, though it may be revised later. Of the three types of domain change requests,
adaptive can be the most difficult to classify. We have found that even though a change may at first appear to be applicable to a domain component, impact analysis may uncover that the change is really an application specific extension and not representative of the requirements of the product family. Why the change is being proposed is as important to making the determination as what the change encompasses. There exists a need for a well defined mechanism to determine overall applicability to the domain. One approach is to try and determine a percentile cut-off for product family representation, before which a requirement is considered application specific. This is particularly problematic as choosing an appropriate cut-off is quite difficult, especially as the cost benefits of the change increase. Another possible approach would be to tie the decision of applicability to overall cost, analyzing the cost of making the change at the domain level vs. the application level. Unfortunately, a focus on short term cost could lead to multiple applications implementing related changes and create long term domain divergence. Regardless of how the determination of applicability is made, the future needs of the individual applications must also be considered in the decision.

The issue of backward compatibility of a change(s) with other fielded applications within the domain is addressed first in the impact analysis step. Here the impact of the change must be assessed for the domain architecture as a whole, as well as the individual impact upon each of the fielded applications (Trace_Impacts in Figure 6). In the ROSE domain maintenance process the domain maintainers traced the impact(s) of the change(s) across the fielded applications. This was possible because the domain maintainers were the domain engineers themselves, who had high familiarity with the applications being developed. This has definite advantages in terms of time required to assess impacts and accuracy of resource estimations, but is probably somewhat exceptional. Another solution would be to add an additional activity to the decomposition of Trace_Impacts where an additional Trace_Impacts method call is made to application maintenance as shown in Figure 7.

The Trace_Impacts method call (1) would then be a request for impact analysis, made of each application maintenance team associated with an already fielded application potentially affected by the changes. This would allow the domain maintenance team to be concerned primarily with building the impact assessment on the domain architecture as a whole and let the application maintenance teams provide specific application costs associated with the change. One disadvantage to this approach is the increased amount of overhead associated with each affected application maintenance team performing an impact assessment, however the final assessment would contain greater accuracy.

6.3 – Cost structure

The use of and Application_Impact_Analysis activity emphasizes the need for a common cost model. Resource estimates would then be generated by adding the resources associated with affecting the change on each fielded application to the resources required for performing the modification to the domain component(s) to arrive at estimated cost for the change. Note that fielded applications would probably not incur these costs immediately because it quite common not to pick up new versions of domain components every time there is a new release.

7 – Application maintenance

As described in section 5.3, the maintenance process is a variation of the generic maintenance process template altered in order to address the three issues introduced earlier.

7.1 – Timing issues

Timing issues are particularly important once interactions between maintenance processes occur. In the case of application maintenance, the impact analysis step includes an explicit interaction with the domain maintainers in order to determine the full impact, cost and feasibility of changes to domain products that would be implied by the resolution of application change requests. Requests with such impacts include, but are not limited to, corrective requests where applica-
tion engineers determined that the problem lies in a domain component, adaptive requests where improvements to the domain architecture would be needed to support the application improvements, etc.

The manner in which impact information is exploited is a process design question with interesting implications. The approach taken for ROSE was to take advantage of the fact that the actual people performing application and domain maintenance are in fact the same. Thus, communication overhead between teams is usually “optimized out” of the process execution. This allows the application maintainers to be empowered with more decision-making power than might be affordable otherwise.

From a more general point of view, a key process design decision is the location of the decision to incorporate domain changes. Application maintainers would like the flexibility to choose whether to make domain changes based upon their cost, but doing so is a costly proposition: it incurs extra delays as detailed information is returned from domain maintenance to the application maintainers and the domain maintainers are notified of the change’s disposition. The former choice is attractive when inter-team communications is rapid and informal while the latter is attractive for its efficiency in more formal settings.

7.2 – Separation of concerns

The location of decision-making is of great importance to the efficient maintenance of applications built using domain components but the two groups’ outlooks drive the decision, as well. Were application maintainers to initially investigate a solution to an application change involving domain changes, yet finally reject it in favor of a local application-level change due to its lower implementation cost or scheduling conflicts, there is the possibility that desirable domain improvements will be overlooked by application maintainers due to their “selfish” view of the application’s maintenance.

It is equally true, however, that domain change requests submitted in the course of analyzing an application change request, when accepted by domain maintainers, can commit application maintainers to a course of action that proves costlier than anticipated.

7.3 – Cost structure

Application and domain maintainers, by being forced to share the same cost model have a much better chance of coming to a mutually agreeable choice when evaluating requests. The cost model needs to include:
- resource estimates to perform requested changes;
- future resource savings estimated following a requested change;
- estimated implementation (calendar) time (a scarce commodity!); and
- if applicable, the cost of maintaining a divergent product set.

The use of a cost vector rather than a scalar value reflects the fact that cost is expressed not only in terms of resources but also time. Furthermore, a more complete cost measure reduces the need to resort to “incen-

tive pricing” by groups that expect benefits out of proportion with the apparent costs (e.g., a domain change that has great long-term benefits but also has a high enough initial cost that the requester, who may only reap part of the benefits, would likely not choose to do it when presented with only resource estimates).

8 – Conclusion

Parnas’ observation that “programs, like people, get old” [19] is doubly true for domain architectures. The impact of “ignorant surgery” is magnified greatly when the patient is a domain component incorporated in numerous fielded applications. We have presented here some of our experience and opinions regarding maintenance of software systems developed in a domain/application engineering paradigm. Our original process models have proven to be robust in a practical domain, but we have also learned some important lessons:

- **Timing / latency:** Detailed diagnostics of where faults lay needed to be done during impact analysis, before making a request for a domain maintenance change in order to help make more time for domain maintenance activities. However, a certain amount of latency seems to be inherent in the relationship between AM and DM.

- **Cost of domain changes:** The costs of making a domain change needed to be taken into account by those requesting it. Alterations to a domain becomes increasingly difficult with many fielded application systems, however beneficial to the overall domain those alterations might be. Decision making requires support for spreadsheet-like “what if” exploration in order to properly understand the trade-offs involved.

**Separation of concerns:** Application and domain maintainers have considerably different outlooks on the products involved. One is narrowly limited to the application while the other is more global. However, both are part of the same organization, and they share a common overall goal.

Parnas’ characterization of retroactive incremental modularization [19] is particularly interesting to consider in this light as a conceptual framework for domain impact modeling. We feel, however, that some form of flexibility and variation must be allowed between domain and application. Negotiated interfaces [22] offer interesting potential here, does as the relative correctness approach of Moriconi and Qian [18].

Acknowledgments

This work has been supported by NASA Cooperative Agreements NCC-9-16 and NCC-9-30, RICIS research activity RB02. The work described in this paper was conducted in collaboration with Rockwell, NASA and Paramax as part of the ROSE reengineering project. The authors are grateful to NASA, UHCL, RICIS and Rockwell for the opportunity to work on this project.

References


