ICS 225 White Paper

The objective of this paper is to provide a *post mortem* of the activites performed in ICS 225, Software Processes. This paper is divided into three sections. The first section will review the effort required to produce the Implementation Workflow of the class project. The second section will discuss the impressions of the author about the class and software processes in general. The third section will consider the future of software processes and software support for processes. While it has been requested that we avoid harping on the shortcomings of Endeavors, there are a number of problems with its philosophy that should be examined, lest the same mistakes be repeated in future attempts to provide workflow automation software.

In terms of effort, it would be easy to say that **at least 90% of the time** allocated to the project was spent attempting to understand and use Endeavors. The **lack of reasonable documentation**, the **inability to save** and exchange work with my partner, and the **excessive size of the Endeavors installation** all contributed to the extraordinary amount of time spent on Endeavors during implementation. The inability to do proper saves was significantly vexing—eventually, I found myself stopping work and copying the entire hierarchy into an archive to preserve my accomplishments so far should the unthinkable happen (a difficult task, as my disk usage was near quota at the time). Endeavors itself has neither a save/restore model nor some form of multi-level undo—any damages done to the workflow currently under design are immediately mirrored to disk; any corruption of the Endeavors database results in **the need to reinstall** a prior archive. This became painfully clear on the eve of the demonstration, where a minor corruption (from hitting the return key, of all things!) resulted in the loss of the previous three hours' work.

I realize that this paper should not be dedicated to Endeavors' shortcomings, but there is one other limitation that should not be ignored: **the complete lack of usable documentation**. Any software that is released undocumented should be immediately regarded with deep suspicion. The lack of documentation indicates that the designers of the system have **no consistent vision** of the purpose of the system, that they have **lost understanding of the goals** behind the development, that they are simply "coding for coding's sake." A live user guide (that is, a user guide that always reflects the current functionality of the system) prevents creeping featurism, obsessing over details and code bloat. It is the *minimum* requirement for releasable software.

In short, **Endeavors is not quality software**; I would be loath to even describe it as pre-beta. I realize that the authors of Endeavors claim that the version used is "just a demo" and does not reflect the commercial-grade package; it has been my experience that while full installations often offer more *features*, they rarely offer greater *functionality.*

The remaining fraction of **the time was divided equally between** understanding and implementing **the USP** and understanding the primary external tool we used, **Rational ROSE**. Both of these efforts were **facilitated by prior experiences**: at Globalstar we used the Booch Method (a variant of the spiral model) and Rational ROSE to develop software, and at KLA–Tencor the pro-

cess book was based upon earlier work by Jacobsen. The need to parse ROSE files and extract relevant information was also required at Globalstar; I applied some of that knowledge to developing the class/module extractor used in the Build Plan generator of the Integrate System step of the Implementation workflow.

In terms of contribution to the entire project, the majority of this effort was wasted—that is, my contribution to the overall workflow was poor. This was not from any conscious action on my part; rather, it was the result of a **lack of communication and numerous misunderstandings** about the activities of others on the project. One shining example of this was the selection of the ROSE document to represent the unified design and deployment model. Since this document had to be shared between the analysis, design, and implementation team and be updated by each workflow with each iteration, it was **our belief that this document should not follow the naming convention** proposed by the data integration team; that is, the document had the possibility of originating from any of the three aforementioned workflows, and would be updated by each. **Trying to agree on a common name proved fruitless**: foreseeing this, we designed our workflow to be able to change the name of the document at runtime, that is, we made the name of the ROSE document an Endeavors field with the idea that when the final document name was agreed upon, we could enter it into the Endeavors application instead of recompiling our Java code.

Another problem had to do with an early **misunderstanding about the nature of the USP workflows**: namely, that each workflow produced or updated some artifact. Unfortunately, the implementation workflow only generates (multiple unnamed) artifacts that make up the Implementation Model: source code, object files, and the like. In fact, the only thing that could even be considered a USP artifact is the Build Plan, which eventually we plan to use as our "signaling artifact" of completion. Normally, the Build Plan is "consumed" by the subsequent implementation tasks.

Early on, my partner and I assumed that everyone would build a **multi-user, multi-tasked** implementation of their workflow; a large portion of the early effort was spent trying to discover how to implement **multiple instances of Endeavors control activity** within a fixed Endeavors workflow. Later on, other groups decided on a **single-user, non-forking** model, and the results of this earlier effort were discarded. We simplified our design with the assumption that at any time there would be *exactly one* active USP workflow; we were surprised when Control Integration announced that they would provide course-grained (high-level) parallelism. Even at this point, parallelism is not well-supported.

Within the Implementation workflow itself, I contributed about **65% of the effort**. This was not a result of a lack of effort on my partner's behalf, but rather because of the natural division of labor: I chose to implement the portions of code that needed to interface with Endeavors, while Sam chose to implement the pop-up dialogs that provided fine-grain control over the activities. The lack of documentation combined with my unfamiliarity with the Java platform made this effort difficult.

While the end result of the project left much to be desired, the actual knowledge gained dur-

ing the implementation was invaluable. A couple of interesting questions have been raised abut the nature of modeling processes in general.

Can a real-world process be modeled accurately with a state-driven workflow? This is a rather pointed question, and a dangerous one depending on who is asked as it could invalidate some pet theories. First, let us refine what we mean by "real-world." A *real-world process* is one that is applicable to the software manufacturing industry; that is, those companies that develop and distribute software for profit. A *state-driven workflow* is one where the workflow consists of a number of discrete steps such that the completion of a step is expected prior to the initiation of the next. State-driven workflows are attractive because it is possible for managers to determine exactly in which state the organization is supposed to be, and where it is supposed to go next. The problem with this naive approach is that software organizations tend not to be uniform collectives; each person within an organization has responsibilities that differ from their coworkers. Additionally, while it is normally impossible for a single worker to perform two tasks simultaneously, it is often the case that when a task becomes locked from unavailable resources, the worker will switch to another task. This kind of **synchronized concurrency is difficult to implement** correctly in state-driven systems without breaking the abstraction. As I will discuss later, implementation of this kind of synchronized concurrency in Endeavors involves constructs that appear to be alien to the Endeavors philosophy.

In general, **are processes state-driven or event-driven?** This could be viewed as a question on the overall marketability of Endeavors and Endeavors-like workflow management systems. The authors of Endeavors ship a sample process—FormDemo—which is supposed to model the approval of expenses at the University of California, Irvine. The FormDemo system works. It works because it effectively follows the life of a **single piece of paper** as it winds its way through the soul-crushing gears of the university's bureaucracy. Forks in that model could be viewed as a quick trip to the copy room; joins as the resounding *thunk* of a stapler. The form itself is ethereal; it exists only as a repository for data (signatures, corrections) picked up along the way. At no point do we actually "see" the form object on the display; we only see its effects. Like a black cat in a dark room, we are only aware of the artifact when it interacts with its environment.

Real processes are unlike that single piece of paper. USP, for instance, involves dozens of artifacts, many of which are modified concurrently by different workers in the process. To model this with automation as a goal would involve a thread of control for each artifact as it is acted upon by each worker. In a highly-tuned organization, no worker would ever be idle, nor would any artifact ever be unused. The end result is that **every task in the workflow would be active simultaneously**; there would be a sea of dancing dots in the Endeavors window, all waiting to proceed to the next step. For all its fancy control-flow graphics, **Endeavors would be converted into an event-driven modeler**.

That's not to say that an event-driven system offers the best presentation of a workflow; far from it. Endeavors (and Endeavors-like systems) have a strong advantage in that **they offer a clear and understandable presentation in terms of steps and goals**. Compare Endeavors to Marvel,

for instance. It is very easy for the human developer to glance at an Endeavors workflow and grasp how many steps will be required to reach a goal, but **in an event-driven system the length of a process is obscured by the notation**. Each stanza represents a mini-task, but the orders in which the stanzas are executed is not known until runtime, and even then the ordering can be changed by the results of prior steps. It might be that the mechanism used to record the process depends on the objective of the process engineer, it might be that **presentation and automation are mutually opposing objectives**.

For systems of limited applicability, this might not be the case. One such system is VOV[1] used in the electronic design world for large-scale automation. Models in VOV are implemented as a restricted form of Petri net—specifically, no cycles are allowed. This means that the graphical presentation of a VOV process is relatively clean (a directed acyclic graph); yet the system is powerful enough to distribute complex tasks across a heterogeneous local-area network. VOV can accomplish what both Endeavors and Marvel fail to (functionality and presentation, respectively) because in its philosophy **it purposely limited its objectives**.

The Endeavors philosophy is limiting as well; whether this is the result of a conscious decision by the designers or a lack of skill by the implementors is unclear. The control-flow system is limited in that it assumes (by its presentation of a "dancing dot") that there will be few active threads of control. The inability to accurately change workflows when the system is active implies that the designers dismissed the need for feedback-driven processes. Support for synchronization of parallel threads of control is lacking beyond the basic 'join' operation. All of these considerations—minor in isolation—combine to make Endeavors unsuitable for anything but the most trivial of tasks.

One could argue that this restricted concurrency model is **the result of the implementation language**. After all, Java only offers primitive thread management capabilities: better synchronization techniques exist in more advanced concurrency languages such as Ada. This is a fallacious argument. Endeavors itself is not married to the implementation language; far from it, considering that the Endeavors object model seems more suited to languages offering a prototype/delegation object model (such as Self, JavaScript, and a number of LISP-based OO dialects). This concurrency model was **a direct decision of the designers** of Endeavors; unfortunately, it is the users of Endeavors that must work around it.

The Endeavors object model is questionable: it consists of three "tiers" of objects: categories, specifications, and instances. Only categories are allowed to have specialized Java code, but all (including user-defined categories) must inherit from a category. The question then becomes: why bother with tiers at all? Simply **offer a flat space** and be done with it! The designers suggested that the three-tiered model was mandated by the implementation but I have seen little to suggest that is so. A flat space would make the mental interpretation easier on both the engineer

1. More information about VOV can be found at `http://www.rtda.com/`. This aside does not constitute an endorsement of the system; I have serious reservations about its fundamental philosophy as well. I only included it as an example of the "middle ground" between Marvel and Endeavors.

and manager, it certainly would make implementation easier, and it may even facilitate a more flexible control-flow model. Alternatively, **use actual Java objects to represent the objects** within the workflow; Java's internal persistence management system is far more flexible and stable than the current system used by Endeavors, and it would make management of custom "categories" easier on the process engineer.

All of these problems with Endeavors could be viewed as a systematic problem in understanding the application domain. The weaknesses described by [GHS95] are all suffered by Endeavors. **Lack of interoperability**? There seems to be no glue allowing the exchange of workflow control between complimentary systems; that responsibility would be up to the user, and would require significant understanding of the Endeavors API. If the other workflow system use a platform-dependent API (shared libraries, perhaps), than the difficulty of providing a bridge doubles because of Endeavors' use of Java's security features. **Lack of support for HAD** (Heterogeneous, Autonomous, and Distributed) systems? Again, the Java platform reduces the end-user to design for the "least common denominator" of operating systems. Fortunately, Java *does* support access to OMG-based ORBs, so that functionality may not be as restricted as non OMG-based workflow management systems. **Inadequate performance for business purposes**? Endeavors speed and functionality is quite limited at this time; it's certainly not ready for "prime time." **Lack of support for correctness and reliability** in the presence of failures? Endeavors offers no form of exception processing that I could see. The default behavior in the presence of a Java exception is to abort that thread of control: not very user-friendly. **Weak tool support for analysis, testing, and debugging**? These features were noticeably absent from this version of Endeavors. On the other hand, the allowed set of workflows provided by Endeavors offer little chance for the more traditional traps of WFMS.

Does all of the blame for the poor results of the class project rest upon Endeavors? **Could not the Unified Software Process be too high-level**, too abstract to implement in a concrete workflow system? Certainly there was a mismatch between the philosophies of the process and the implementation toolkit, but this mismatch is not a result of a limitation in either system. While the USP has problems, these problems only marginally impacted the implementation of the USP in Endeavors.

One of the problems with the USP is that **it does not scale down** easily; that is, it makes fundamental assumptions about the size of the organization that implements it. One could argue that it would be possible in a smaller organization to have some of the workers play more than one role in the USP—merging the system architect and system integrator roles is a natural consideration. This scaling, however, can only go so far: at a point, certain roles become redundant, even detrimental, to software development. Certainly at the single-user level most of the roles become unnecessary, so **implementing the USP as a single-person effort was a ineffectual undertaking**. That the early constraints of modeling a single-person, non-concurrent process in Endeavors meant we were implementing something fundamentally against the USP philosophy.

One idea not brought up in class but mentioned frequently in lab was the feeling that the

USP was implementing something that was already there. Many of these roles—system architect, test engineer, *et cetera*—existed far earlier than the USP and its ilk. One could view Jacobson *et al*. as simply **giving names to things people already know**. While this as a valid interpretation of the USP, it is not necessarily a shortcoming. One problem with process development—even informal development—is the communication of ideas to others. One of the first tasks of a development group should be to come up with a glossary of terms and to stick with it. By naming things people "already know" **it becomes possible for workers to express dissatisfaction** with the existing process and suggest improvements that can be understood by other stakeholders. (As an aside, that is why [FH93] will have a subtle but long-lasting impact on the software process community than any other paper on process engineering.)

One metric could be to see how a process improvement technique such as the **Capability Maturity Model "rates" a hypothetical organization fully implementing the USP**. A white paper published by Rational[2] on the Rational Unified Process (a superset of the USP) states that a proper implementation of their process will support all of the key process areas required for Level-2 (Repeatable) and Level-3 (Defined).

For instance, the **Requirements Management KPA** of Level-2 (which experience has shown to be the hardest for an immature organization to achieve) consists of two goals: 1) that software requirements are controlled to establish a baseline, and 2) that software artifacts and activities are kept consistent with the requirements baseline. The Rational white paper states that the four major milestones—Lifecycle Objectives, Lifecycle Architecture, Initial Operational Capability, and Product Release—make up the requirements baseline. This is a reasonable declaration, as all other artifacts in the USP are derived from or contribute to those milestones. The second goal, consistency, is a little harder to close. Since the USP encapsulates requirements as use cases, and the use cases are used to derive products of the latter workflows, the end result is that all software artifacts are consistent with the requirements. This assumes that the "translation" from use case to operational software is controlled.

On the other hand, **many Level-3 KPAs are not covered by the USP** but are covered by the Rational Unified Process. Peer review is a good example. There is no mention of peer review in the USP book; which is unusual since the effectiveness of peer review has been shown to be invaluable especially in the Implementation (coding) work of a project [PSTV97]. The Rational Unified Process *does* mention peer review in a number of cases. Why was this critical portion of a software development process left out of the USP? Perhaps the author did not believe in peer review's effectiveness. Alternatively, it is possible that Jacobsen felt that peer review was tied more closely to the instantiated process rather than a generalized process, and that it would be transparently incorporated into the various tasks where it is required (code development, use-case refinement, *et cetera*). It is hard to see where code review would be incorporated into the "Implement a Class" task—is it part of the implementation of operations, or does it fall under the generalized sub-

2. `http://www.rational.com/products/rup/prodinfo/whitepapers/dynamic.jtmpl?doc_key=100416`

task of ensuring that the component provides the same interfaces as its corresponding design class? Nor is it clear where—if at all—it would appear in the Test workflow. Perhaps in a future edition of the USP handbook Jacobsen will handle this significant omission.

While the white paper was enlightening, there is the danger that **it is heavily biased** by the author's association with the system and thus lacks credibility; from experience, I know of one case where a software process management vendor stated—without proof—that implementing their techniques would automatically move the organization to "CMM Level 2.3 (*sic*)."[3]

All in all, most of the problems encountered during the implementation were either the result of implementation errors in Endeavors or **philosophical conflicts between Endeavors and the USP**. As mentioned before, Endeavors best models a state-driven system with limited concurrency, while the USP requires a document-driven approach with significant concurrency. How would a system reconcile these different approaches?

One way Endeavors could be made to model this is to implement **a mechanism which allows threads of control to terminate gently**.[4] The idea here is to provide a series of prerequisite checks prior to instigating a task. Should the prerequisites fail, the thread of control associated with that pending task would be terminated. The idea here is that one could model an arbitrary boolean prerequisite as a network of singleton guardian conditions prior to execution of a task.

Of course, one could take this approach to the extreme. By providing two incoming ports and two outgoing ports on every task, **any arbitrary Horn clause could be modeled**; basically, Endeavors would become a form of visual Prolog. The result would be a conversion from a simple state-driven control-flow system to a full rule-based event system. The only problem would be that the activity performed would be obscured by the complexity of the resulting network; in effect, the problem that besieges Marvel and other rule-based process management software.

Additionally, one could argue that a significant portion of the implementation problems resulted from taking **too high of an abstraction of the USP**. Our original workflow design was hierarchical, with each of the five major tasks (Architectural Implementation, System Integration, Subsystem Implementation, Class Implementation, and Unit Testing) having its own workflow and sub-tasks. When it became apparent that time and complexity restrictions were going to make this approach infeasible, we opted to eliminate all of the subtasks by representing each task with a single action. As the model was refined to a single-user, non-parallel solution, we further reduced the size of the workflow from five tasks to four, consolidating the Subsystem Implementation and Class Implementation tasks into a single task, since the two were only differentiated to facilitate parallelism within the USP. We simulated this parallelism by providing a mechanism to iterate over the build plan produced by the System Integration task. The result was something that **only superficially represented the USP**. Examination of other core workflow implementations showed similar disappointing results.

---

3. Joint Unisys/IBM FSC development group, Sunnyvale, California, 1993.

4. This mechanism may already exist. The lack of user documentation and the limited time prevented a full investigation into this capability.

Was time, then, the actual problem? One could argue that point. But the time restrictions only accentuated the existing problems with Endeavors—lack of documentation and inherent instability. Additional time may have allowed the core workflow developers to analyze and reverse-engineer Endeavors, getting a better idea of which capabilities it had and whether or not it could implement some of the more complex control flow scenarios mentioned above. On the other hand, documentation—any documentation at all—would have alleviated the need for such experiments.

Perhaps the real killer of the project was **the lack of a class-wide system architect with a clear vision of the goals of the implementation**. Early on, the professor should have appointed a student to lead the project or should have taken on that role himself. It was the lack of a common goal that lead to incompatible implementations. One could even argue that **it is absurd to try to implement a process without using a process**. The approach used by the class was very *ad hoc*: each group did its own thing with the hope that at the end each module could be seamlessly integrated into a functional whole.

The actual benefit of the project might very well be not a better understanding of the Unified Software Process, but **a better understanding of the need for a process under any collaborative effort**. Many of the students in the class have never worked in a "real-world" company; those few with work experience have seen the industry through the chaotic haze of a start-up company.

It might be **impossible to design a workflow management system that can be used for any arbitrary process**; indeed, there seems to be no common idea of the functionality needed even when discussing the same process. In our class different teams had different ideas of what the goals of the project were. While I envisioned a system highly automated and massively parallel, others in the class envisioned something simpler, decomposable, and maintainable. These conflicting visions led to conflicting implementations based upon conflicting assumptions about the other implementations and, ultimately, a much more difficult period of integration.

In general, it seems that **we can divide workflow management tools into three groups**: those that are **so highly specialized that the cannot be applied to all workflow problems** (*make*, VOV), those that are **so abstract as to make it impossible to maintain** all but the simplest workflows (Marvel), and those **that provide too simple an abstraction** as to make all but the most primitive workflows impossible to implement (Endeavors).

make and its ilk were designed for a very specific purpose, not generalized workflow; this is the reason behind their power, flexibility, and widespread acceptance. While these tools have often been used for applications for which they were never designed, the normal situation is that they are used for the task to which they are best suited. *make* builds software from the bottom up: it is based upon the simple idea that when a task (a sequence of build commands) is complete, the product's timestamp will be newer than those items used in its manufacture. But it is a workflow tool: it streamlines a process by generalizing the steps needed in the construction of hierarchical software. Tools exist to display *make*'s build hierarchy in a graphical format, and tools exist even to automatically and non-intrusively covert user actions into a *make*-like build process: a primitive form of process discovery (VOV).

Marvel, on the other hand, has the **extreme opposite philosophy: it tries to do everything**, in such a way that any process can be described by it. Unfortunately, the external representation that Marvel uses—rules that respond to events, both external and internal—makes the problem of process management all the more severe. Gone is the ability to "at a glance" understand the high-level nature of the process. One must be willing to read in every rule, understand those events that said rule responds to and generates, then be able to place the rules in a semblance of an order, hoping that when the "process" is running no external events interfere and change the linear model as it is understood. The inability to understand a system without copious external documentation limits the ability of a defined process to survive a multi-generation project where the implementor of the project might be replaced midway through with a person unfamiliar with "the way things are done." The lack of an easily-understood presentation model limits the usefulness of Marvel.

Endeavors makes things too simple. Not all processes are sequential tasks with deterministic branches. Most processes model more than one person, performing disjoint tasks in parallel. While the presentation of the process is nice and clean with levels of hierarchy allowing the process engineer to refine his knowledge as the need arises, the consequence is that the underlying model must also follow the same restrictive limitations. As we determined in the class project, implementation of a process involving multiple actors working on multiple artifacts in multiple independent tasks is difficult, near impossible, to implement and still provide the clean presentation that makes Endeavors attractive from the start.

The problems with current process management systems are a direct result of the belief that process mechanics are well-understood and can be cleanly abstracted. Understanding the need for parallelism—explicit and implicit—would make Endeavors a better product. Understanding the need for a clean presentation would have made Marvel a more usable system. Either a good abstraction for event-driven models or a better method of implementing concurrent tasks in state-driven models needs to be developed; until then, **process management software will be nothing more than a curiosity**, relegated to the dusty back shelf with all other software purchased for "show."

While the software is hurting, **software processes are alive and well**. Some, like the USP, provide a formal nomenclature for the corporate procedures already underway. Others attempt to discover processes through careful observation and instrumentation. Still others attempt to mechanize error location and optimization through formal notations reminiscent of programming languages. On the pragmatic side, major corporations are adapting formal verification and auditing metrics to determine the "health" of their processes. While engineers may balk at this culture change, it is important that they realize software has reached a level of complexity which makes it nearly impossible for a single person to fully understand a system—the need for coordination, control, and quality mandates a process.