

Software Process Modelling Example for ISPW-7 Call for Participation

15 August 1991

1 Background

As part of the Seventh International Software Process Workshop (ISPW-7), participants are invited to solve a process modelling problem, to submit those solutions for examination, and to meet on the day before the workshop proper to discuss those solutions. The purpose of this exercise is to enhance our mutual understanding of the variety of approaches that have been developed for software process modeling.

As many of you know, the idea of providing a modelling problem was first attempted last year as part of ISPW-6. This exercise proved to be very successful and everyone felt that it should be continued in future workshops. Rather than trying to construct a completely new problem, the ISPW-7 problem extends and modified the ISPW-6 problem. The extensions attempt to address issues in two areas: team work and process change.

2 Instructions

The example problem consists of a required core problem and several optional extensions. For those of you who did not participate last year, you may choose to just do the core part of the problem. You should make a good faith effort to comprehensively solve the entire core problem. As a result of the ISPW-6 exercise, it is clear that most approaches can expect to encounter some aspects of the core problem that either cannot be expressed or are quite awkward to handle. It would be quite helpful in such cases if the modeler would identify these difficulties in comments accompanying the solution.

For those of you who have seen the problem before, we have provided some suggested modifications to the problem and we would expect you to address one or more of these problems. As with the new-comers, we would hope that you will provide some written insight into your solution and the problems that you encountered.

3 Distribution of Solutions

Solutions to the example problem will be distributed to participants prior to the Workshop. Electronic distribution will be used where possible. This will enable people to review them in advance, thus speeding our efforts during the working session. The Software Design and Analysis company (SDA) has agreed to duplicate and distribute the solutions. Incidentally, this will be a separate mailing from the position paper preprints.

Example problem solutions must be received by *October 1, 1991* in order to be duplicated and distributed. Solutions may be sent in *Postscript* form via e-mail, or in plain *ASCII* text via e-mail. Paper copies may also be sent, but E-mail is preferred for fast distribution. *No Other Formats Are Acceptable. Do not send*

material with embedded formatting commands that must be submitted to a text processor! Send solutions to

ISPW-7 Software Design and Analysis POB 3521 Boulder, CO 80303 USA e-mail: ispw7@sda.com

If your solution is not ready by the October 1 deadline, you are, of course, free to distribute your solution at the example session. But there is no guarantee that you will be given time for a formal presentation.

4 Special Working Session

A special working session to review the solutions is planned for Tuesday, 15 October 1991, in the same Hotel as the workshop. It is anticipated that this will be a full-day session.

This year, unlike last year, all participants of the workshop are invited to attend the example session to be held on the day before the workshop. You need not submit a solution to attend and discuss the solutions. However, formal presentations at the example session will be limited to those participants who have submitted a solution.

On the day of the example session, we will divide into two or three groups each of which will discuss their solutions. The basis of the division into groups is still tentative. It will depend on the number and nature of the submitted solutions. At the moment we have two proposals for dividing into groups:

1. Divide by nature of the modelling language used. In this case, we might expect to divide into two basic groups: those using a non-executable process modelling language and those using an executable process programming language. This latter group might in turn be divided into one group whose languages are primarily procedural and another group whose language is primarily rule-based.
2. Divide by solution. Those whose solution focused on team issues would be in one group and those who focused on process change in another group. New-comers would be divided equally among the two groups.

Each group will have a designated leader who will be responsible for guiding the presentations and discussions. Each group will be expected to provide a summary of their discussions. We intend to provide some specific questions for each groups to answer in its summary. During the regular workshop, one session will be devoted to presentations of the group summaries. Additional details regarding the meeting will be distributed as they become available.

5 Concluding Notes

This modified solution is the result of efforts by Marc Kellner, Dewayne Perry, the program committee of ISPW-7, and, of course, the ISPW-6 group responsible for the original problem. The members of the ISPW-7 committee sincerely hope that you will participate in this exploration of modeling approaches, and we look forward to meeting with you at the workshop.

Following this cover letter are three sections. First is a description of the teamwork extension. Second is the description of the process change extension. Third is the text of the original ISPW-6 problem statement.

Marc I. Kellner
Software Engineering Institute
and
Dennis Heimburger
University of Colorado, Boulder

ISPW-7 Problem

(Extensions to ISPW-6 Problem)

The set of new extensions to the ISPW-6 example has been divided into two parts. One part focuses on teamwork aspects and a second part focuses on process change. In turn, each part has several specific extensions. You should choose to solve one of the specific extensions under one part. Thus, you may choose one of 4.1.1, 4.1.2, 4.1.3, 4.2.1, or 4.2.2 to solve. You are, of course, free to do more than one part or to add detail (within reason) to your choice if you feel it will show useful features of your formalism.

4.1 Teamwork

The teamwork extensions actually consist of two extensions plus a collation of the teamwork-related options from the original example statement. The extensions address two specific issues in teamwork. One issue (4.1.1) is the managerial problem of allocating various constrained resources, especially human resources, in order to complete a schedule of tasks. A second issue (4.1.2) is the coordination of multiple programmers working in parallel.

Section 4.1.3 is slightly different. The observation has been made that the original example, and especially the optional extensions, contained a significant amount of teamwork related exercises; it is especially rich in role-related material. As part of this exercise, we decided to collate much of the pre-existing material and allow it to serve as an additional “extension” to be solved. If you choose to use section 4.1.3, please note carefully exactly which parts you are addressing. If you have not attempted the ISPW-6 example before, you may find that solving one or more of the pieces mentioned in 4.1.3 can help focus your solution to the whole problem.

4.1.1 Resource Management

Extend the information available about a schedule to include duration and personnel information. Assume a set of reasonable durations for each step in the process (e.g., 3 days to modify design, 4 days to modify code, etc.). Based on your process model, attempt to generate a schedule for the completion of the process. Now, given this schedule and a resource database about personnel commitments, attempt to assign people to tasks in accordance with commitment and time constraints. Specifically address the problem of people serving multiple roles. You may simplify the problem if you wish by assuming that the schedule to be addressed has no loops in it. That is, it represents a linear sequence of steps through the process assuming perfect execution of all steps (e.g., the modified design is approved at the first review).

The intent of this choice is to explore the management issues involved in process scheduling, resource management, and role assignments under constraints. It is also intended to exercise the data modeling capabilities of your approach.

4.1.2 Coordination and Communication

This extension involves converting the code modification step (2.7) of the original example from a single person activity to an activity involving parallel modifications and merging of effort. Assume that multiple, interdependent pieces of code must be chosen and modified to satisfy the design modification. Given the pieces of software, your solution should model the instantiation of parallel modification steps (possibly including assignment of people to each parallel activity). Your solution should also allow for interdependencies between code pieces and for coordination of changes between interdependent code pieces. As a concrete scenario to consider, assume that two pieces of code have been chosen for modification and that they are to be modified in parallel. But assume that the two pieces of code are interdependent. For example, assume that one piece defines an interface used by the other and that this interface needs to be changed. The two programmers involved should notify each other of the required change, should perhaps review each other’s changes, and should coordinate to decide when the two pieces are finished and consistent.

The intent of this choice is to exercise your ability to specify concurrent activities and to demonstrate how, concretely, team members will coordinate activities. There are some dimensions to this problem that may not be immediately apparent. The first dimension is the degree of coupling between coordinating participants. In this problem we assume the coordination is what we will term “asynchronous,” as opposed to “synchronous.” By synchronous coordination, we mean to imply that the people doing the coordinating are typically in the same room and will primarily coordinate by direct verbal communication. Part 2.6 of the original example would be an instance of synchronous coordination. In asynchronous coordination, we assume that the participants mostly operate independently and when they coordinate, they use the process formalism as an intermediary for that coordination.

The second dimension is the dynamic specification of the number of parallel activities. Obviously if one specifies the number of parallel threads and their interdependencies in advance, then it is relatively easy to accommodate them in any reasonable formalism. The intent here is that the number of pieces of code that must be modified is determined as part of the process and then an appropriate number of threads of activity are created. It is these threads that must coordinate with each other.

4.1.3 Options from the Original Example

The ISPW-6 Software Process Example (original problem) illustrates a number of teamwork issues in various ways throughout its structure. At a high level of abstraction, it assigns a group responsibility for a high level process abstraction. That is then decomposed into more refined and specific tasks and associated responsibilities. The original example also details many specifics of the interactions between activities and agents (flows of information and objects, as inputs and outputs; physical communication mechanisms for conveying those flows; coordination details specified as constraints on sequencing, pre-conditions, etc.). The issue of multiple roles is also a thread throughout the existing core problem. Three original extensions provide additional semi-structured opportunities to explore teamwork issues more fully.

More specifically, Develop Change and Test Unit (2.3 - the overall process) is carried out by the project team (2.3.4), whose structure is spelled out in the last paragraph of 2.2. The original example goes on to decompose this higher level process into 8 major components, each of which has its own assignment of personnel responsible for performing that task. Two of these major components are specifically carried out by teams. The Review Design step (2.6) is carried out by a team of 4 specified individuals (2.6.4), including the design and QA engineers working this change plus two other software engineers. Similarly, Test Unit (2.10) is carried out by a two-person team of the design and QA engineers working this change (2.10.4).

The original example also illustrates role issues. Specifically, the key individuals (agents) in this process (i.e., project manager, specific design engineer, specific QA engineer) each play multiple roles within the process. For example, the QA engineer has four roles: (a) reviewer (2.6 - Review Design), (b) test plan modifier (2.8 - Modify Test Plans), (c) unit test package modifier (2.9 - Modify Unit Test Package), (d) tester (2.10 - Test Unit).

Three of the original extensions provide additional settings within which to explore teamwork issues:

- Review Design Details (3.3)
- Scheduling and Resource Constraints (3.4), and especially Scheduling Under Resource Constraints (3.4.3)
- Aggregation of Multiple Changes (3.5), especially with interacting changes

Finally, we should note that the new extensions above (4.1.1, and 4.1.2) essentially elaborate the original extensions 3.4.3 and 3.5, respectively, to focus on team work issues.

4.2 Process Change

The process change extensions addresses two forms of change. One, which we term “process modification,” is intended to represent some form of permanent evolution of a process. The second form of process change,

which we term “process exception,” concerns a temporary modification to the process to handle some exceptional circumstance.

Note that the mechanisms involved in handling either part may be very similar and it would be of interest if you explained why (or why not) your approach for one extension would differ for the other.

4.2.1 Process Modification

In the original example, it was possible to begin coding before design was approved. Suppose that it is decided to change the process and to tighten the requirements detailing when coding can begin (2.7.4.1) in order to require that the design be approved before coding may begin. Further, assume that this process change is to affect some currently “executing” or “instantiated” process. Show how the change affects the state of that executing process and how the consequences of the change will be communicated to the user of the process. You may want to consider three cases of process modification.

1. The executing process has not yet reached the step affected by the change, so the change will have no immediate impact on the process state. In this case, that would mean that coding has not yet begun. Note: this case is sometimes termed a “static” process change.
2. The executing process has reached or passed the steps affected by the change, but for whatever reason, the change is consistent with the existing process state. In this case, that would mean that some coding was in progress, but it just happened that no coding actually occurred before the design was approved.
3. The executing process has reached or passed the steps affected by the change, and the change is inconsistent with the existing process state. In this case, that would mean that some coding was in progress but the design was not yet approved. Note that you have two options here: (1) you may allow the inconsistency to remain, or (2) you may attempt to change the state (by rollback, or whatever) to achieve consistency.

The intent of this extension is to highlight the mechanism by which various approaches handle changes in an already defined process model and to begin to address some of the issues involved in on-the-fly modification of processes.

4.2.2 Process Exceptions

Suppose that due to unavailability of assigned personnel, it is decided to bypass a follow-up (say second) design review. This decision is made dynamically at the time that this review was scheduled to occur. The rest of the process continues normally in this instance. How would the process model cope with such a dynamic change? Note that in this scenario, we would expect you to minimize the undesirable side-effects of the exception. For instance, if later steps in the process assume (and check for) completion of all design reviews, then we would hope that handling the original exception would not also produce an exceptional condition at these later steps.

The intent of this choice is to explore how your approach handles exceptional situations: situations that were not covered in the original process, but which are not viewed as requiring permanent changes in the process model.

ISPW-6 SOFTWARE PROCESS EXAMPLE

Marc I. Kellner, Peter H. Feiler,
Anthony Finkelstein, Takuya Katayama,
Leon J. Osterweil, Maria H. Penedo,
and H. Dieter Rombach

28 August 1991

Note

This software process example was originally published in the “Proceedings of the 6th International Software Process Workshop: Support for the Software Process” (Hakodate, Hokkaido, Japan; 28-31 October, 1990; edited by Takuya Katayama; published by IEEE Computer Society Press, 1991), under the title “Software Process Modeling Example Problem”. The title has been changed to “ISPW-6 Software Process Example” in order to readily distinguish this original example and version from subsequent ones.

Preface

One of the important activities undertaken in conjunction with the 6th International Software Process Workshop (ISPW-6) was a comparison of various solutions to a standard software process modeling example problem. The primary purpose of this effort was to facilitate understanding, comparing, and assessing the various approaches that are being pursued for software process modeling. Secondary goals included communicating the diversity of aspects of process actually encountered in modeling real-world software processes, and providing impetus to efforts to extend the proposed approaches to cope effectively with these process issues.

It was determined that these objectives would be best served by soliciting solutions to a standard software process modeling example problem. The use of a standard benchmark problem facilitates comparisons of various modeling approaches. Consequently, a working group was formed in 1990 to devise the example problem. This working group consisted of the following individuals:

- Marc I. Kellner – Chairperson (Software Engineering Institute; Pittsburgh, PA, USA)
- Peter H. Feiler (Software Engineering Institute; Pittsburgh, PA, USA)
- Anthony Finkelstein (Imperial College; London, UK)
- Takuya Katayama (Tokyo Institute of Technology; Tokyo, Japan)
- Leon J. Osterweil (University of California at Irvine; Irvine, CA, USA)
- Maria H. Penedo (TRW; Redondo Beach, CA, USA)
- H. Dieter Rombach (University of Maryland; College Park, MD, USA)

This group has devised a problem that comprehensively exercises the various modeling approaches being developed, through coverage of several important components of real-world software processes. The full problem is presented below, as it was distributed to ISPW-6 participants.

1 Introduction

This example software process modeling problem has been designed to aid in understanding and comparing various approaches to software process modeling. In making such comparisons, one is usually drawn to

an evaluation of the relative strengths and weaknesses of the approaches under examination. However, it should be recognized that a determination of strengths and weaknesses must be based upon some set of goals and objectives which a given approach is intended to achieve. For example, direct executability is of paramount importance for an approach whose major goal is to provide automated support for process enactment; on the other hand, it may be relatively inconsequential for an approach focused upon facilitating human understanding and communication regarding process.

In order to facilitate understanding and comparisons, this problem has been painstakingly designed to contain a large number of different types of process issues seen in real-world software processes. This provides an opportunity to demonstrate the capability to model well over a dozen different categories of process issues.

This example problem consists of a core problem and several optional extensions. Solution of the core problem is required, in order to provide a common ground for beginning to understand different modeling approaches. The optional extensions provide additional opportunities to demonstrate the capabilities of different approaches, and are much more open-ended.

We have striven to strike a difficult balance between a small, simple problem and a rich, complex problem. The former extreme becomes relatively useless and unrealistic, while the latter extreme becomes unwieldy and overwhelming. We have also chosen to draw the example problem from the software process domain, and not from a “foreign” domain, to ensure relevance. In addition, we have opted for a careful specification of the core problem, rather than leaving it either to our common understanding or open-ended. While this makes the problem development and presentation much more lengthy, it is expected to provide a firm, consistent basis for comparison of the solutions. The optional extensions complement this with ample opportunity for showcasing other capabilities.

2 Core Problem Description

2.1 Overview

The core problem is scoped as a relatively confined portion of the software change process. It focuses on the designing, coding, unit testing, and management of a rather localized change to a software system. This is prompted by a change in requirements, and can be thought of as occurring either late in the development phase or during the support (maintenance and enhancements) phase of the life-cycle. In order to confine the problem, we assume that the proposed requirements change has already been analyzed and approved by the Configuration Control Board (CCB) or other appropriate authority. It has been determined that only a single code unit (e.g., module) is to be affected. The problem begins with the project manager scheduling the change and assigning the work to appropriate staff. The example problem ends when the new version of the code has successfully passed the new unit tests.

Some method had to be determined for describing this problem. Textual narrative offers the advantage of being widely understood and was selected. Still, some organization of the narrative is required to make the problem statement relatively comprehensible. It was decided to organize the narrative by major task, and to modestly structure each task description. While it is recognized that any organization may bias solutions in a particular direction, this organization was felt to offer minimal bias while still enhancing understandability.

The narrative for each task begins with an overall description of the specific step. It would be desirable to demonstrate that such narrative descriptions can be somehow handled (at least recorded) in a modeling approach. Following the description, lists of inputs and of outputs are provided. The source is listed for each input, and the destination is listed for each output. The physical communication mechanism (e.g., e-mail, or hand carried) is also indicated. It is noteworthy that all possible inputs and outputs are listed in these sections, and it is entirely possible that not all occur in any given instance of the task. The particulars are detailed in the description section; for example, “design review feedback” is an output of review design only if the design is not approved. The party responsible for carrying out the task is indicated next. After that, various constraints regarding the sequencing of this step are described. Finally, some tasks also include another section containing additional information, such as regarding the details of the objects being manipulated.

The first description is of an encapsulating abstraction of this entire example process (just the core problem), entitled Develop Change and Test Unit. Following that, the component steps of this example process are described. They are

- Schedule and Assign Tasks
- Modify Design
- Review Design
- Modify Code
- Modify Test Plans
- Modify Unit Test Package
- Test Unit
- Monitor Progress

Preceding these descriptions, however, is a brief section discussing some global issues.

2.2 Global Issues

Assume that there are no resource constraints or conflicts in this core problem. Consequently, work on the change can begin as soon as the engineers receive their assignments. No delays are encountered due to unavailability of resources.

For the sake of brevity, certain common inputs and outputs have not been listed exhaustively. Define “technical steps” as those not performed by the project manager. The following information is available to every technical step in this process; consequently, they are not listed for each step individually

- Requirements change (from schedule and assign tasks) (hand carried)
- Notification of task assignments and scheduled dates (from schedule and assign tasks) (e-mail)
- Notification of revised task assignments and scheduled dates (from monitor progress) (e-mail)

In addition, the following information is provided by each technical step in this process; consequently, they are not listed for each step individually

- Notification of completion of each step (to monitor progress) (e-mail)

Feedback to earlier steps is to be modeled only when explicitly indicated, such as from review design back to modify design. While it is recognized that many other instances of such feedback occur in practice (e.g., from modify code back to modify design or even back to clarify requirements, or from test unit back to modify design), they need not be modeled in this example problem. Our interest here is in determining how a given modeling approach deals with such feedback, not in exhaustively modeling the full richness of a real process.

For the sake of variety, some objects (e.g., designs) are assumed to be represented manually on paper, while others (e.g., source code) are represented on computer media. The specification of physical communication mechanisms (or media) was in some cases rather arbitrary. It is permissible to substitute an alternative for any of the manual instances (i.e., one that is either verbal or hand carried). For example, one may wish to indicate that designs are in automated form rather than paper. However, automated cases (i.e., e-mail or computer I/O) are to remain as specified. Furthermore, a number of objects in this problem are described as being under automated configuration management. These include source code, object code, and unit test packages. In such cases, it is intended that the solutions will show extraction of the latest version of the object for modification, and later insertion of a new version.

Each solution of this core problem should show the encapsulating abstraction (Develop Change and Test Unit) and, ultimately, its decomposition into the eight component steps described herein. However, intermediate groupings or abstractions may be devised and employed as desired; for instance, one might wish to use the grouping “technical steps” defined above.

The organizational structure relevant to this example process will be described here. A project team is responsible for work on the software system under consideration. It consists of a project manager and a group of software engineers. The software engineering staff contains a group of “design engineers” (whose primary responsibilities are design and coding) and a group of “quality assurance (QA) engineers” (whose primary responsibilities are test development and execution). Specific responsibility for tasks will be described below. In addition, a Configuration Control Board (CCB), outside the project team, provides overall guidance and authorizations.

2.3 Develop Change and Test Unit

2.3.1 Description

This step is a high-level abstraction of the process described in this core problem. The details are described in the remainder of the core problem description. It is intended that solutions show this high-level abstraction as well as its decomposition into the details.

2.3.2 Inputs

1. Requirements change (from CCB) (hand carried)
2. Authorization (from CCB) (verbal)

2.3.3 Outputs

1. No explicit output flow. The final results are available in various files or databases and comprise the modified software unit in various representations (design, source code, object code) and its associated modified unit tests. These are all available to subsequent steps in the software life cycle.

2.3.4 Responsibility

This step is carried out by the project team.

2.3.5 Constraints

1. This step can begin as soon as authorization to make the change has been granted by the CCB.
2. This step ends when the unit testing has been successfully completed, or when the CCB cancels the change effort.

2.4 Schedule and Assign Tasks

2.4.1 Description

This step is a project management function, and is the first step carried out in this example process. It involves developing a schedule for the work to be undertaken for this software change, and assigning individual tasks to specific staff members: specifically, a design engineer and a quality assurance (QA) engineer. The design review is also scheduled, and participants are assigned to it; the participants will be the design and QA engineer, along with two other software engineers.

Assume that the basic process to be followed in this example is already familiar to the participants; it is standard procedure for this organization. Consequently, this management step focuses on assignment of resources to the tasks and estimation of the schedule required for those resources to carry out the tasks.

2.4.2 Inputs

1. Requirements change (from CCB) (hand carried)
2. Authorization (from CCB) (verbal)
3. Project plans (from file) (computer I/O)

2.4.3 Outputs

1. Updated project plans (to file) (computer I/O)
2. Notification of task assignments and scheduled dates (to all affected personnel) (e-mail)
3. Requirements change (to all assigned personnel) (hand carried)

2.4.4 Responsibility

This step is carried out by the project manager.

2.4.5 Constraints

1. This step can begin as soon as authorization to make the change is granted by the CCB.
2. This step ends when its outputs have been provided; assume that all outputs are produced simultaneously.

2.5 Modify Design

2.5.1 Description

This step involves the modification of the design for the code unit affected by the requirements change. It is a highly creative task. The modified design will be reviewed, and ultimately implemented in code. This step may also modify the design based upon feedback from the design review.

2.5.2 Inputs

1. Current design (from software design document file) (hand carried)
2. Design review feedback (from design review) (hand carried)

2.5.3 Outputs

1. Modified design (to review design, modify code, modify unit test package) (hand carried)

2.5.4 Responsibility

This step is carried out by the assigned design engineer.

2.5.5 Constraints

1. This step can begin as soon as the task has been assigned by the project manager.
2. Subsequent iterations can begin as soon as the design review is completed (when the design is not approved).
3. This step ends when its output has been provided.

2.6 Review Design

2.6.1 Description

This step involves the formal review of the modified design. It is conducted by a team including the design engineer who produced the design modifications. There are three alternative outcomes of the review:

1. Unconditional approval – The design is totally approved; the approved modified design is incorporated into the software design document.
2. Minor changes recommended – Minor changes to the design are required and feedback is provided to the designer. The re-review is expected to be perfunctory.
3. Major changes recommended – Major changes to the design are required and feedback is provided to the designer.

At the conclusion of the review, the project manager is notified of the outcome. Due to the fact that formal design reviews are a relatively new step in this organization's process, they are recording certain measurements to help evaluate its impact. In particular, the number of defects identified, and the aggregate effort of the review team in preparing for and conducting the review are reported to the project manager.

2.6.2 Inputs

1. Modified design (from modify design) (hand carried)

2.6.3 Outputs

1. Design review feedback (to modify design) (hand carried)
2. Approved modified design (to software design document file) (hand carried)
3. Outcome notification, number of defects identified, aggregate effort (to monitor progress) (e-mail)

2.6.4 Responsibility

This step is carried out by the design review team assigned by the project manager. This team includes the design engineer, QA engineer, and two other software engineers.

2.6.5 Constraints

1. This step will be carried out when it is scheduled to occur, provided that the modified design is available at that time. (Note that in cases of delay, the monitor progress step will reschedule the review to a later date.)
2. This step ends when its outputs have been provided; assume that all outputs are produced simultaneously.

2.7 Modify Code

2.7.1 Inputs

This step involves the implementation of the design changes into code, and compilation of the modified source code into object code. Implementation is accomplished by modifying existing source code. This step may also be based upon feedback from testing, indicating that additional source code modifications are required.

2.7.2 Inputs

1. Modified design (from modify design) (hand carried)
2. Current source code (from software development files) (computer I/O)
3. Feedback regarding code (from test unit) (verbal)

2.7.3 Outputs

1. Modified source code (to software development files) (computer I/O)
2. Object code (to software development files) (computer I/O)

2.7.4 Responsibility

This step is carried out by the design engineer.

2.7.5 Constraints

1. This step can begin as soon as the task has been assigned by the project manager. (Thus, it is possible, if not advisable, to begin coding before the design work has even begun.) The engineer's discretion determines when this task will actually commence, if the design has not yet been approved. If this step has not commenced earlier, it will begin when the design is approved.
2. Modified code released for testing is to reflect the approved modified design. One consequence is that this step cannot be completed before the modified design has been approved.
3. This step ends when a clean compilation (no errors) has been accomplished and its output has been provided.
4. Subsequent iterations (if required) can begin as soon as the test unit step has completed.

2.7.6 Additional Information

This step deals with a set of objects (entities) whose attributes and interrelationships need to be recorded. Both source and object code are under automated configuration management. The latest version of the software unit's source code provides the starting point for modification during this process step. Upon successful compilation, a new version of the unit's source code is recorded, along with its corresponding object code. In addition, these must be related to the modified design upon which they are based.

Source code information includes the body, creation timestamp, version information, and engineer responsible for this change. Object code information includes the body, target system, and creation timestamp. Each source code instance can have multiple corresponding object code instances. Similarly each design version can relate to multiple source code instances. For each successful compilation of a given source code instance, it is important to record the timestamp, the compiler version used, all compiler options selected (e.g., optimizer on, debugging on), any error or warning messages produced, and listings and associated documentation such as memory maps.

2.8 Modify Test Plans

2.8.1 Description

This step involves the modification of test plans and objectives to include testing of capabilities related to the requirements change prompting this software modification. These test plans are analogous to software designs. They identify the functionality and capabilities to be tested, and the approach to be taken. On the

other hand, the specifics of actual test data, procedures, and so forth are dealt with in the subsequent step (modify unit test package).

2.8.2 Inputs

1. Current test plans (from test plans file) (hand carried)

2.8.3 Outputs

1. Modified test plans (to test plans file) (hand carried)

2.8.4 Responsibility

This step is carried out by the assigned QA engineer.

2.8.5 Constraints

1. This step can begin as soon as the task has been assigned by the project manager.
2. This step ends when its output has been provided.

2.9 Modify Unit Test Package

2.9.1 Description

This step involves the modification of the actual unit test package for the affected code unit, in accordance with the modifications made to the test plans and objectives. It will be assumed that the test package contains computer software to drive and evaluate the unit under test, along with narrative procedures recorded in accompanying computer files. These test packages are under automated configuration management, and a new version of the unit test package is created when this step ends. Finally, it is noted that the modified design and/or the source code for this unit may be used as input to this step; however, this is not always the case.

Subsequent iterations of this step may be based upon feedback from testing, indicating that additional modifications to the unit test package are required.

2.9.2 Inputs

1. Modified test plans (from test plans file) (hand carried)
2. Current unit test package (from test package file) (computer I/O)
3. Modified design (from modify design) (hand carried)
4. Source code (from software development files) (computer I/O)
5. Feedback regarding test package (from test unit) (verbal)

2.9.3 Outputs

1. Modified unit test package (to test package file) (computer I/O)

2.9.4 Responsibility

This step is carried out by the assigned QA engineer.

2.9.5 Constraints

1. This step can begin as soon as the modify test plans step has completed.
2. This step ends when its output has been produced.
3. Subsequent iterations (if required) can begin as soon as the test unit step has completed.

2.9.6 Additional Information

Each instance of the unit test package includes software, procedures, creation timestamp, version information, and engineer responsible for this package instance.

2.10 Test Unit

2.10.1 Description

This step involves the application of the unit test package on the modified code, and analysis of the results. Both the design and QA engineers participate in this step. The entire test package is run before any analysis or further action is taken. Although the unit tests are primarily functional, an automated coverage analyzer is employed to determine that adequate test coverage of the unit's code has been achieved; a 90% successfully completed, and 90% successfully passed. When this is not the case, the design and QA engineers jointly analyze the test results and determine appropriate actions. In order to confine the problem, assume that either or both of the following are the outcomes of this analysis: the source code needs to be modified further, the unit tests need to be modified further. When the indicated modifications have been made, the unit can be retested.

If the unit passes its tests and meets an acceptable coverage level, then this example process has been completed. Steps beyond unit testing, such as integration testing, are beyond the scope of the core problem. The project manager is notified that the latest version of the unit code is available for integration testing.

2.10.2 Inputs

1. Object code (from software development files) (computer I/O)
2. Unit test package (from test package file) (computer I/O)

2.10.3 Outputs

1. Test results (to test history file) (computer I/O)
2. Feedback regarding code (to modify code) (verbal)
3. Feedback regarding test package (to modify unit test package) (verbal)
4. Notification of successful testing (to monitor progress) (e-mail)

2.10.4 Responsibility

This step is carried out jointly by the design and QA engineers.

2.10.5 Constraints

1. This step can begin as soon as both the object code and unit test package are available.
2. This step ends when its outputs have been provided; assume that all outputs are produced simultaneously.

2.10.6 Additional Information

Each instance of the execution of a set of unit tests upon an instance of the code is recorded in the unit test history file. This includes identification of the unit test package version and object code version used, along with the timestamp of the test execution, indication of tests failed, and the coverage level attained. Note the possibility that, over time, multiple unit test package versions may be applied to a given object code version, and vice versa.

2.11 Monitor Progress

2.11.1 Description

This step involves the project manager monitoring progress and status of the work. This is based upon notification of completion of each step, as well as informal information. While work proceeds according to plan, no action is taken and no output is developed by this step. On the other hand, deviations from plan can result in rescheduling of tasks. In severe cases, the project manager may put the entire modification effort on hold, and recommend to the CCB that this change effort be canceled. The CCB may either concur and cancel the effort, or indicate that it is to be resumed where it left off. In such cases the affected personnel are notified (by the project manager) as appropriate regarding the resumption or cancellation of their work on this change.

2.11.2 Inputs

1. Notification of completion (from all technical tasks) (via e-mail)
2. Current project plans (from file) (computer I/O)
3. Outcome notification, number of defects identified, aggregate effort (from review design) (e-mail)
4. Notification of successful testing (from test unit) (e-mail)
5. Decision regarding cancellation (from CCB) (verbal)

2.11.3 Outputs

1. Updated project plans (to file) (computer I/O)
2. Notification of revised task assignments and scheduled dates (to all affected personnel) (e-mail)
3. Recommendation to cancel (to CCB) (verbal)

2.11.4 Responsibility

This step is carried out by the project manager.

2.11.5 Constraints

1. This step can begin as soon as the initial task, schedule and assign tasks, is completed.
2. This step persists throughout the duration of this process. For the purposes of this example, assume that this task ends
 - (a) when the unit testing has been successfully completed, or
 - (b) after notifying personnel of cancellation by the CCB.

3 Optional Extensions

3.1 Overview

This section presents a number of optional extensions to the core problem described above. Their purpose is to provide a variety of opportunities for showcasing capabilities of a given modeling approach that may not be demonstrated by the core problem. Each extension builds upon the core problem, and they can be considered independently of each other. They are rather open-ended, leaving considerable freedom to orient them as desired.

3.2 Modify Code Details

This extension entails developing the details of the modify code step more fully. Direct modifications are made to the source code by the engineer, while object code is produced by a compiler. The current source code (latest version) is withdrawn from the configuration management library at the outset of this step. The engineer invokes an editor to make his changes to the source code. When he decides that the source code is ready for compilation, he saves the source code file and invokes the compiler on it. If there are compilation errors, he goes back to the source code to make additional modifications and then recompile. If a clean compilation has been achieved before the design has been approved, a final manual check of the code is made by the engineer after the design is approved and before the code is released for unit testing; this may result in additional source code changes.

It may be convenient (although not required) to assume that intermediate versions of the code objects and compilation information and products are maintained in a private library (database) under configuration (version) management. Only the final version needs to be placed into the software development files.

3.2.1 Basic Elaboration of Details

One option is to demonstrate handling of low level process steps and details by representing the above description, adding additional elaboration as appropriate.

3.2.2 Automatic Tool Invocation

A more ambitious extension would combine the above with automatic invocation of automated tools. These should include at least the editor and compiler. It would also be desirable to include automatic connections to an assumed configuration management tool, and perhaps automatic generation of the e-mail notification message when the final code version is placed in the software development file (making it available for unit testing).

3.3 Review Design Details

This extension entails developing the details of the review design step more fully. This step is one carried out by a group of engineers, and involves considerable inter-personal communication, collaboration, and cooperation. Further elaboration of this step would deal with the dynamics of how such a group review process occurs. Any reasonable design review procedures may be assumed to incorporate this extension.

3.4 Scheduling and Resource Constraints

This extension deals with the project management issues of scheduling and resource constraints.

3.4.1 Basic Scheduling

Assume a set of reasonable durations for each step in the process (e.g., 3 days to modify design, 4 days to modify code, etc.). Assume that resources are available as needed to accomplish the tasks without delay. Assumptions must also be made to handle the iteration loops in this process (such as cycling through the modify design - review design loop a few times until the design is approved). Indicate how an initial schedule (before the work has begun) would be determined based upon the process model. Incidentally, this work goes on within the schedule and assign tasks step.

3.4.2 Schedule Revision

Indicate how the initial schedule, derived above, would be modified to cope with actual results obtained during process enactment. For example, assume that coding actually takes 2 days longer than anticipated. What is the impact on the revised schedule for completion of unit testing, and how is this determined? Incidentally, this work goes on within the monitor progress step.

3.4.3 Scheduling Under Resource Constraints

Modify one or both of the preceding scheduling extensions to cope with resource constraints. These might include unavailability of personnel when a task is ready to be started, or partial availability (such as 50% can be devoted to the task).

3.5 Aggregation of Multiple Changes

This extension entails the consideration of multiple changes. Rather than considering only one change to the software, assume that an arbitrary number are being developed independently, and possibly concurrently. The process for each individual change is as described in the core problem. However, when the full set of changes have passed their independent unit tests, they are integrated together into a full system build and integration testing is conducted on this new developmental configuration. Of course, integration testing may reveal problems in the units changed, or elsewhere in the system, and corresponding additional modifications will have to be made to correct these.

3.6 Process Change

This extension involves identifying and implementing changes to the process. The easier case would involve analyzing the process to identify opportunities for improvement, selecting one or more, and putting them into practice in the future. An example of this would include analyzing the defects identified during unit testing, and trying to improve future design reviews to catch those types of errors. This specific example would involve adding additional information to the data recorded as a result of testing (such as to classify the errors), and a specific post-mortem step oriented toward process improvement.

A more challenging case would involve dynamic process change during execution. For example, suppose that due to unavailability of assigned personnel, it is decided to bypass a follow-up (say second) design review. This decision is made dynamically at the time that this review was scheduled to occur. The rest of the process continues in this instance. How would the process model cope with such a dynamic change?

3.7 Product Representation

This extension provides an opportunity to examine the relationships between a process modeling approach and the underlying product representation. Assume a specific implementation and/or design language for the software product. Then develop the process model in sufficient detail to illustrate the process acting upon the product (considering the product to be represented in the chosen language). Identify and examine the ways in which the product representation language impacts the process model and the software process modeling approach.

3.8 Modeler's Choice

Those solving the problem are invited to include any other extensions that illustrate special capabilities of their modeling approaches. The nature of each problem extension should be explained, and the demonstrated special capabilities should be highlighted.