# Findings and Recommendations for a Software Development Process

**Intelligent Systems Division**

Elena Messina
David Coombs
Tom Kramer
John Michaloski
Fred Proctor
Will Shackleford
Keith Stouffer
Tsung-Ming Tsai

# Table of Contents

## Abstract

Researchers in the Intelligent Systems Division (ISD) at the National Institute of Standards and Technology frequently develop software-intensive advanced prototype systems with applications in manufacturing and defense. Given the reliance on software for the majority of the programs within this division, it is critical that the software be of high quality and effective. The Software Development Process Project was initiated within ISD to assess the current state of software development practices and to chart a course for improvement. This paper summarizes the findings of the team that was assembled to perform this investigation. Issues discussed range from development tools and infrastructure to division-wide software life cycle process definition. A brief set of recommendations is presented at the conclusion.

**Keywords**: software development, life cycle, software process

## 1.0 Introduction

Software development is a key activity within the Intelligent Systems Division (ISD) of the Manufacturing Engineering Laboratory at the National Institute of Standards and Technology (NIST). Researchers in this organization develop software-intensive advanced systems with a variety of applications for industry and defense. Given the reliance on software for the majority of the programs within this division, it is critical that the software developed be high quality and effective.

The Software Development Process Project (SDPP or SDP) was initiated within ISD to assess the current state of software development practices and to chart a course for improvement. A team was formed with representatives from various groups and projects to present a balanced picture of the various facets of software development within the division. This document presents the findings of the group and suggests follow-on activities.

The document begins with some definitions of concepts that will be used in later sections. The major findings regarding the current state of ISD software development are then enumerated. Each one of these points is scrutinized. Appropriate actions are proposed, along with the potential benefits. The next sections discuss in depth the issue of a software development process model. A brief description of the software process models adopted by Sandia National Laboratories is included.This provides a benchmark and an example of a process model in use. A proposed process model for software development in ISD is then formed. The elements comprising the model are described and project categories are proposed. The relationship of the proposed process model to the ISD project management model is clarified. The document concludes with a section summarizing the overall short-term recommendations for the division. As an exercise in imagination which unifies a lot of the points covered in this document, a vision for ISD software development a few years hence is submitted as an appendix.

## 2.0 Goals

The goals guiding the Software Development Process Project team were the following:

- Improve the understanding of the software development processes and requirements within ISD
- Develop software process model(s) for the division which improve the efficiency and quality of projects
- Begin deploying methodologies and tools to support software processes and to improve efficiency and quality of projects

The project was undertaken to discover how software is currently developed and to determine perceived bottlenecks or weaknesses in the current approach.

## 3.0 Scope

The SDP Project investigated typical software activities by project teams. The project did not attempt to assess project development issues, such as scheduling, requirements analysis, or resources. Hardware issues were not discussed, unless they affected software development. The discussions were biased toward “research” types of applications, although some consideration was given to software that is distributed to other organizations.

The type of software development activities performed within ISD are not generally discussed by the software engineering community. Most of the focus for software process discussions and studies within the software engineering community is based on the assumption that some type of “product” is being built. We typically build prototype systems. We aim to demonstrate or test some underlying algorithm or principle, rather than provide a practically defect-free piece of software with a flashy Graphical User Interface (GUI) to our paying customers. This, of course, is an exaggeration. However, the “center of gravity” for our projects is certainly skewed towards prototype demonstration systems.

Another aspect that complicates this endeavor is the nature of our organization. By education, experience, and career interest, most of the researchers who write software are not software engineers, nor is this a full-time activity for most of them. The ISD organization’s modest size and finite budget also bound our options for change.

## 4.0 Some Software Engineering Definitions

There are many perspectives to a software-based endeavor. We describe some of the major perspectives of “software” in this section and distinguish between them.

### 4.1 Software Engineering

Software Engineering is establishing and using sound engineering principles to obtain economical software that is reliable and works efficiently on real machines [1]. Software Engineering encompasses methods, tools, and procedures to provide a *paradigm* to enable developers to build high-quality systems and to enable managers to achieve control over the development [2].

### 4.2 Software Methodology, also known as Software Methods

Software methodology provides the technical foundation for the software. Methodology is an organized, documented set of procedures and guidelines for one or more phases of the software life cycle, such as analysis or design. Many methodologies include a diagramming notation for documenting the results of the procedure; a step-by-step approach for carrying out the procedure; and an objective (ideally quantified) set of criteria for determining whether or not the results of the procedure are of acceptable quality.

## 4.3 Software Process, also known as Software Procedures

Software procedures define the following: the sequence in which methods will be applied, the deliverables that are required, the controls that help ensure quality and coordinate change, and the milestones that enable software managers to assess progress [2].

## 4.4 Software Tools

A software tool is a program primarily used to create, manipulate, modify, or analyze other programs, such as a compiler, an editor or a cross-referencing program. Tools provide automated or semi-automated support for methods.

## 4.5 Software Development Environment

The software development environment refers to the infrastructure supporting the mechanics of building, testing, and updating software, particularly when several people collaborate. Intuitively, the complexity of an environment which supports multiple developers working concurrently on the same software system is greater than for stand-alone, individual efforts.

## 4.6 Configuration and Code Management

Configuration management entails controlling the releases of and changes to software items throughout the life-cycle. Similarly, a source code management system helps program developers track version history, releases, and parallel versions.

## 4.7 Quality Assurance

Quality assurance refers to the procedures in place to ensure that a system meets the requirements. The measures of quality are correctness, maintainability, integrity, and usability. If quality is measured, traditionally, it is measured after a system is built. A more modern approach involves considering all or most aspects of quality throughout the life cycle.

## 4.8 Life Cycle of Software

A discussion of process typically includes a description of a software life-cycle model. A software life-cycle model describes the stages and procedures involved in developing a software system. They include the following:

- System engineering and analysis - probe and establish requirements for all system elements; allocate some subset of the requirements to software; perform top-level design and analysis
- Software requirements analysis - focusing on software elements, establish the requirements; understand the information, function, performance, and interfaces; document these and review with “customer.”

- Design - translate requirements into a representation of the software that can be assessed for quality before coding begins; document the data structures, software architecture, procedural details, and interfaces.
- Coding - translate design into executable form
- Testing - ensure that program behavior agrees with expected, correct results
- Maintenance - modify an existing software system to correct defects, or to accommodate new requirements.

The classic life-cycle model is the waterfall, shown in Figure 1. This model prescribes sequential flow through the various stages of software development listed above. Although there is some provision for feedback, this model does not adequately allow for modifications based on new information gained throughout the development. Errors inserted in earlier phases may not be discovered until later ones, when it is more difficult and expensive to correct them.

An important element for any life-cycle model is the inclusion of explicit criteria for exiting one phase and entering the next. For example, a criterion would be a clear statement requiring the completion of design reviews for all modules prior to progressing from the “design” to the “coding” phase. Such a statement defines the distinct project phases and the agreed-upon conventions for completion of a phase. Defined criteria make assessing and reporting the status of a project somewhat clearer. Compare a statement such as “We have completed the design for 25 out of 35 modules” to “We are about half done with the project.” The former statement more clearly defines

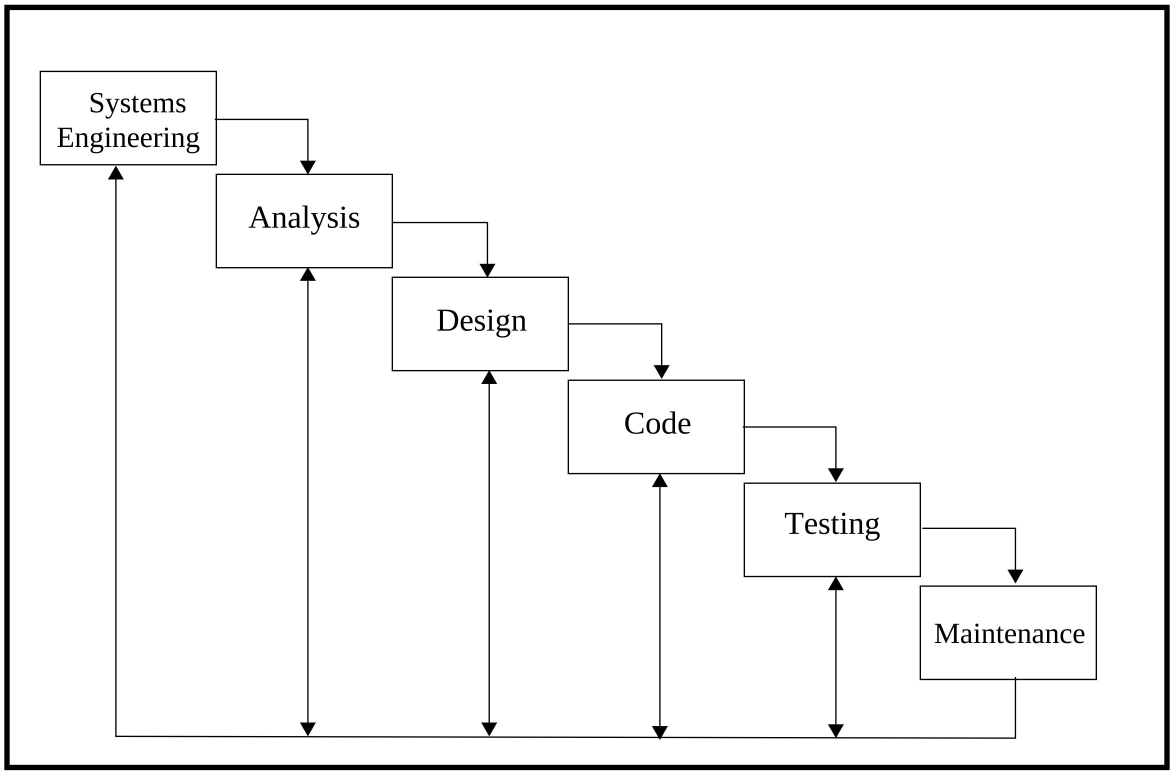

**Figure 1: The Waterfall, or Classic Life Cycle**

the status; the latter is qualitative. Project managers who have many years of experience have encountered projects whose status seems to stabilize at 80% done. A finer granularity for reporting would help pinpoint just where the bottlenecks occur [3].

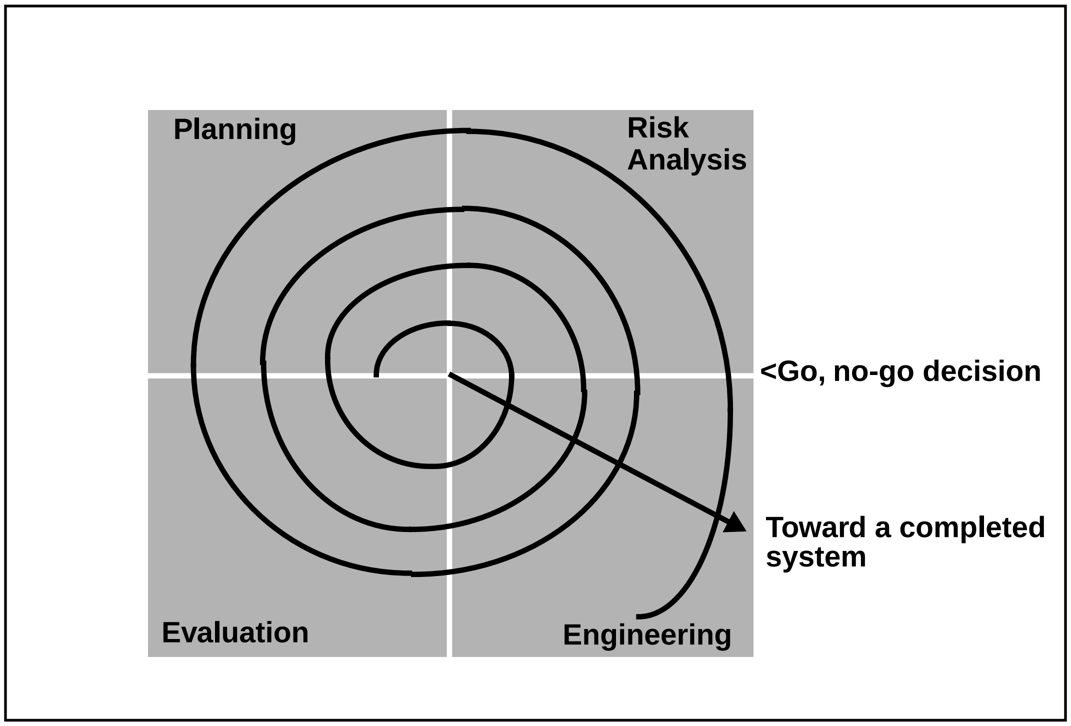

**Figure 2: The Spiral Life Cycle**

In 1988, Boehm introduced a new model for representing software development called the Spiral Life Cycle [4]. It is illustrated in Figure 2. This model supports prototyping to uncover issues early and to allow for customer and management evaluation before too much resource is committed. The spiral model also provides a more explicit phase for assessing the current state of the project. The Boehm view of software engineering iterates through four phases, each time moving the project closer to the desired final state. The four phases are: planning, risk analysis, engineering, and customer evaluation. They are described below:

- Planning - determining objectives, alternatives, and constraints. The initial planning encompasses first-pass requirements gathering. In later iterations, planning takes into account new data, such as schedule slippage or customer feedback.
- Risk Analysis - dissecting alternatives and identifying or resolving risks. In later iterations, this is based on new data, similarly to the planning phase. At the end of this phase, the risks and their potential for resolution are assessed. A decision is made whether to continue with the next iteration of the project or not (“go, no-go”).
- Engineering - incorporating the latest constraints, refining the product design and implementation.
- Customer evaluation - assessing the results of engineering and reviewing new design, schedule, prototype, or other outcome.

This approach is attractive because a project is not allowed to move forward indefinitely before managers and developers discover that it has strayed beyond the customer expectations or budgetary constraints. The “build a little - learn a little” approach may also encourage forward movement.

## 5.0 Findings and Proposed Responses

To determine the amorphous issue of assessing “software development” within ISD, the team held wide-ranging discussions to ascertain the key issues. Topics were suggested as discussion items. If a topic proved to be fruitful, follow-up discussions or action items resulted.

The discussions led to a set of conclusions about software development practices and needs. The list below enumerates these conclusions. Further amplification is provided in following sections.

### 5.1 ISD currently follows an ad-hoc process.

**Discussion:** There are no stated guidelines to define stages in a software project or determining progress. For example, there are no guidelines to decide when to use prototyping. There is no standard software engineering methodology or process in place for a project to follow.

**Proposal:** ISD should adopt a project classification system and associated process models. Projects should be assigned one of the decided-upon broad categories, based on the defined discrimination factors. Each level or category has a process model that a project will adopt. The key factor is management support for the importance of adhering to a process. Studies show that “Senior Management must believe software process improvement is both achievable and necessary and must insist on performance. It is not enough for Senior Management to just say they support software process improvement.” [5] What typically happens is that, despite everyone’s best intentions, “when push comes to shove, the [project] work wins out and the process improvement often dies.” [5] A proposed ISD model is described in Section 7.0.

**Benefits:** If certain guidelines are provided, the decision process can be streamlined to determine the appropriate course a project should follow. Expectations are clarified, and better communication can result. Duplication of effort should also be reduced by ensuring that broadly-applicable tools are leveraged across projects. Elucidating options and available techniques or tools will help disseminate best practices that currently reside in “islands of knowledge.” The proposed processes are intended to be evolutionary and not dogmatic. Flexibility is desirable, as is adaptability based on trial and error.

### 5.2 C++ and a unix-based model for configuration management and for development are “standard.”

**Discussion:** The convergence upon a single implementation language should simplify developers’ lives and make some of the desired changes, such as reuse, more feasible.

**Proposal:** This is an acceptable status quo for now. The trend in industry is toward PC-based systems. So ISD must consider how to support a heterogeneous development environment at a future date.

### 5.3 Tools to help real-time system development are needed.

**Discussion:** There is a lack of software tools within the division. Although tools provide no magic, they make a difference in productivity and quality. Examples of tools that would be beneficial include state diagram and state table generation utilities, timing tools, modeling tools, and communications tools that generate and automatically test communication channels.

**Proposal:** The Systems Team currently provides support via built-in capabilities that are available with the Communication Management System (CMS) package. Additional requests for tools or utilities supporting real-time code development are being collected. A “buy-or-build” decision can be made for each item. A simple process for submitting requests or ideas to the Systems Team currently exists: informal verbal requests are made to representatives of the Systems Team. If the number of “customers requests” increases sufficiently to justify more formality, a simple e-mail request system can be established to log and track the requirements. The systems team should also provide the division with regular updates on planned enhancements to their tools. Periodic mailings to the isd-tech distribution list would be appropriate.

**Benefits:** Development efforts can be reduced by having available tools to support real-time systems. Reliability of systems can be strengthened by using tools addressing real-time software development. Metrics on the performance of systems would be achievable. Having the Systems team be the focal point for these tools would eliminate duplicate efforts and build focused expertise.

### 5.4 Knowledge and experience are not effectively leveraged.

**Discussion:** Expertise is scattered throughout ISD. Knowledge gained with certain tools or subsystems is not necessarily shared with others who could benefit. This is not a malicious withholding of information. Even in our process discussions, sharing of experiences occurred. There is no standard forum for sharing experiences, or even for developers to be aware of someone else’s experience in a given area.

**Proposal:** The ISD internal World-Wide Web page is a good start toward a focal point for information sharing. The team members suggest better use of e-mail queries. A newsgroup for ISD technical discussions is an alternative, but it is not recommended, because it would have very low traffic, and developers are reluctant to have to launch a newsreader application.

**Benefits:** Information-sharing can lead to more effective implementation and dissemination of best practices. Airing questions and issues can also indicate which areas of software, tools, or even theory need more attention. For instance, if a particular utility draws a lot of questions, this may indicate that it either needs better documentation or a better interface. It may also be defective.

### 5.5 Better documentation is needed on all fronts.

**Discussion:** A desire for better communication creates the need to document utility libraries, such as matrix math, that people commonly use. We also need better documentation of code that we develop and others could use. Beyond just module documentation, developers also need examples of how to use or build systems, like the Enhanced Machine Controller (EMC).

**Proposal:** More focus is being given to documentation. RCS documentation is now available on the ISD Web page. This is a big step forward.

Additional documentation issues include:

- An on-line listing of all software available in the division (or available from other sources, such as MSID) should be developed and maintained.

- On-line documentation should be searchable.
- With standardization of module/function header comments, automated documentation extraction is possible.
- We should experiment with tools which generate c++ class documentation in HTML format.

**Benefits:** On-line listing of existing utilities and tools can make software development more efficient and allow for the reuse of code.

## 5.6 The development environment is challenging.

**Discussion:** It is complicated to set up a new project's development environment (e.g., make files), even if it builds upon an existing one. Beyond that, design, development, and integration mechanics are difficult when there are several developers who must coordinate their activities.

**Proposal:** New documentation on how to build RCS systems will help. The Systems Team will take a stronger role in providing application development environments. They are looking at modifying the development environment mechanics:

- use the Revision Control System code management system, which is newer and believed to be better than Source Code Control System (SCCS)
- have the developer's local environment be able to reference object libraries in centralized locations, simplifying the "make" maintenance.
- provide better means of "checkpointing" releases (see Item 5.7 below).

**Benefits:** Developers perceive that too much time is spent figuring out how to build a system. Any improvements in this area can increase the efficiency of the organization. The reworked environment should facilitate development of systems by several researchers who need to work on related code.

## 5.7 Code configuration management needs to be more flexible.

**Discussion:** Developers need better configuration control to be able to get back to a known, stable point. Currently, complications arise if one has to provide a customer with a correction or update on an older version of software. Better configuration management tools should help with notification of changes, fixes, new releases of RCS or other shared software.

**Proposal:** There are several concerns regarding configuration management. The adoption of Revision Control System code management may facilitate addressing these concerns. Additional recommended considerations for configuration management design are:

- Understand and enumerate the various possible states a project can be in (in development, integration, final testing, initial release-release N).
- Understand and enumerate the various possible states in which a developer's local environment can be (e.g., totally self-contained within a single developer's environment; single developer has some local code, references some code in a centralized location; multiple developers share some local code; multiple developers share some local code, but reference some code in a centralized location).
- Understand and enumerate the platform configurations.
- Determine the expected longevity of a project. Will code be shipped outside NIST? If so, we may need to maintain it at a future date. If so, we need to track the various releases and archive the source and objects.

- Once the variables are enumerated and understood, a rational plan for which directories need to exist and their inter-relationships can be developed. It is desirable to limit the proliferation of configuration management models.

**Benefits:** A revamped configuration management system can allow checkpointing of systems more readily. Earlier, working versions of a system can be retrieved. An appropriate directory structure can facilitate the integration and collaboration among developers. Projects with simpler needs should not be burdened with a lot of complexity in the directories and configurations.

## 5.8 The definition of Quality Assurance is currently vague.

**Discussion:** Within ISD, Quality Assurance typically consists of testing at the end of development. The extent and means of testing are left up to the individual. New systems must be quality assured (especially if delivered to outside parties), and software which is used within others' systems needs to be stable. Hence, if a CMS function changes, existing systems should not break.

**Proposal:** Quality cannot be tested into a system. It is much higher payoff to be conscious of quality throughout the entire lifecycle of a project. The software process should define the kind of quality assurance that is required for a given project category. Better use can be made of utilities such as code coverage analyzers and memory leak detectors. Design reviews and use of existing components can ameliorate the situation as well.
The process will encourage projects to begin testing integrated components as soon as possible. Following the "build-a-little, test-a-little" paradigm, early integration testing is advocated. A formal, rigorous definition of quality criteria is not currently attempted.

**Benefits:** Introducing quality assurance earlier in the development cycle results in cost savings for the division. By having a framework of quality assurance tools and techniques, developers will have concrete options in validating their systems. Everyone in the division can have access to the same "toolkit" of techniques and tools and share expertise.

## 5.9 There is a tendency to get into RCS theory discussions when reviewing software design

**Discussion:** A clearer understanding of the difference between RCS architecture and purely software design is needed.

**Proposal:** The division's awareness of the difference between the RCS reference architecture and the software design, which addresses implementation, is growing. We need to continue going in this direction. The HPCC Reference Architecture Project will solidify RCS to a great extent and provide a more concrete reference architecture upon which to build. That should reduce the RCS theory discussions when implementation issues are being presented.

**Benefits:** By focusing on software design which follows an accepted RCS theory, meetings would be more productive. In fact, having a strong reference architecture with some supporting tools should lead to faster progress and better quality.

## 5.10 It is challenging to reuse existing code.

**Discussion:** For as many systems as ISD builds, very little software is reused. Perceived factors reducing the ability to reuse software are: platform portability complexities, lack of documentation for existing software, and software is not built with "generality" in mind. Machine-dependent assumptions may be buried throughout the code. This makes it difficult to locate all such instances and modify them to support another configuration or machine.

**Proposal:** Several initiatives are already in place to increase reuse. Stressing the benefits of reuse throughout the organization is necessary to reap the rewards. A fair amount of discipline is required to successfully implement reuse.
Areas in which the Systems Team is helping to promote reuse:

- better documentation of existing utilities
- more portable code
- better support via the proposed process

The HPCC Reference Architecture project is developing an architecture, both in the conceptual and engineering sense, which can be leveraged for reuse. Note that reuse need not apply solely to existing code. Reuse of designs, data structures, and objects provides significant benefits as well.

As new systems are built under the new regime, reuse should be encouraged and tracked. The software process should make this a natural part of doing business. The ISD project reviews should include explicit queries regarding reuse. The project review team should include at least one individual capable of evaluating reuse opportunities.

Portability-friendly practices should be publicized. Greater awareness by developers of general things to avoid, such as hard-coding degrees-of-freedom, can lead to more reusability.

Browsing existing software is an important aid to portability. A web search for certain keywords, for example, should be possible.

**Benefits:** Reusability benefits the organization because it reduces the amount of development necessary in a new project. Reusing a module that already has been exercised in one or more systems already increases the reliability of the system.

### 5.11 An "endorsed," or "suggested" toolset is desirable, as long as support for these tools is available.

**Discussion:** Currently, projects independently investigate available tools and either decide to purchase a tool or do without. The decision is often made in isolation from other activities, past, present, or future. The outcome is a proliferation of similar tools, or a compromised result, caused by purchasing the cheapest option - or none at all - due to insufficient funds. One example cited is a GUI-building toolset.

**Proposal:** Certain individuals or groups should be given explicit responsibility for supporting tools which are used division-wide. This means that they will be allowed budget and resources to provide the support. At this time, we do not specify individuals or groups, except to mention that the Systems Team already plays this role for certain development tools, like gnu and Centerline. Similar support should be provided for CAD, GUI, and other tools.

**Benefits:** Centralizing the expertise reduces the overall cost of support to the organization. Supported tools also have a better chance of being more thoroughly and widely used.

### 5.12 Different types of software projects require different "rules" to live by.

**Discussion:** Due to the varied nature of the projects in the division, it would be counterproductive to decree a "one-size-fits-all" process model. See section 7.0.

**Proposal:** We propose a multi-tiered approach to project classification. Each tier will have a process that imposes a level of rigor appropriate to its scope and nature.

**Benefits:** By not endorsing a single process model, there is a greater chance for acceptance and adoption within the organization. Demonstrating understanding of the organization's multiple facets increases confidence in the viability of the recommendations. Tailoring process models to the needs of a project is also advantageous. The intention is to enhance the overall effectiveness of the project teams, not merely introduce process for its own sake.

### 5.13 Any recommendations need to consider the research nature of our work.

**Discussion:** Again, it would be counterproductive to impose too much rigor into our environment. Too much energy would be expended on areas where there would be minimal returns for the project's goals. See item 5.12.

## 6.0 The Sandia Software Development Process Models

### 6.1 Looking at What Another Lab Does

Sandia National Laboratories initiated an effort to investigate and define the Software Development Process Methodologies used within their organizations [6]. A team of software developers from various organizations regularly met to produce and promote a core set of software development processes. A detailed description of the process methodologies was produced for the overall laboratory. The Intelligent Systems and Robotics Center (ISRC) developed a specification for all software development activities within that division, referencing the Sandia-wide methods. Because this division is roughly our counterpart at another federal research lab, we looked at the process components and categorization used by Sandia's Intelligent Systems and Robotics Center.

Conformance to the Process is addressed by having adherence to the guidelines in every manager's performance agreement.

**Table 1: Sandia Software Process Categories**

| Rigor | Category | Name |
|---|---|---|
| highest | I | production |
| . | II | prototype |
| . | III | research |
| lowest | IV | rapid response |

### 6.2 Sandia Process Categories

The categories Sandia used are listed in Table 1 and are defined as follows:

I. Production: Software that will be distributed to external agencies, to the public domain, or distributed where a software failure could have real or perceptual consequences. Example:

DOE pit handling systems intended for production use, which are part of the Nuclear Weapons Complex projects.

II. Prototype: Software developed for proof-of-concept systems, projects with evolving requirements, and test-bed systems. The consequences of a software failure are not critical.

III. Research: Software in a project where requirements and/or design issues are ambiguous.

IV. Rapid Response: software that is quickly developed to meet short time schedule for demonstration purposes only. Such software is expected to have a limited life.

There is a Software Toolbox developed by ISRC, probably similar to our RCS tools. The Toolbox can contain software from any of these categories. However, a project may not use Toolbox software from a less rigorous category.

The level of rigor applied to a given software category directs which documents and reviews are required for a given project. In some cases, the decision is made by the individual project teams.

## 6.3 Sandia Documents

The elemental documents that a project may have are listed and briefly described:

**Software Requirements Specification(SRS)** - WHAT the software must do. The appropriate level of rigor and content for this specification depends on the particular project.

**Operator Interface Specification** - describes how an operator interacts with a system.

**Software Design Description** - describes the major components of the software design, i.e., HOW the software will work. The choice of methodology and format is determined by the project leader.

**Software Test Plan** - describes the specific test cases that will be used to ensure that the code, when executed, complies with the requirements specified in the Software Requirements Specification.

**Software Test Results Summary** - documents that the test plan and script were followed. If any differences or deviations from expected results were found, they must be identified in a defects report.

**User Manuals** - depending on the nature of the project, these can be one or more of the following: Operator's Manual, Programmers's Manual, and Software Maintenance Manual.

**Software Project Plan** - this plan, developed prior to the start of software development, identities tasks, and major milestones.

**Software Configuration Management Plan (SCMP)** - identifies configuration management requirements that are unique to a given project, causing a deviation from the center-wide global configuration management plan.

## 6.4 Sandia Reviews

The types of reviews possibly required for a project are listed and briefly described in this section. Apparently, the software project team attends all reviews and invites customers to certain appropriate reviews. There is no discussion of how or when external reviewers are selected.

**Software Requirements Specification Review** - held to ensure that the system's requirements are correctly and completely translated to the SRS.

**Operator Interface Specification Review** - held to verify that the customer agrees with the operator interface design.

**Strategic Design Review** - is held to verify that the project is consistent with the strategic intent of the center. This high-level review, where the primary audience consists of representatives from the Software Process Team, is used to determine several things: (1) whether the project will reuse existing center and/or commercial software where appropriate, (2) if the project has commonality with other projects where collaboration may be useful, and (3) to identify portions of a project that should be targeted for the ISRC Software Toolbox.

**Software Design Description Review** - verifies that the requirements in the Software Requirements Specification are implemented in the design expressed in the Software Design Document. One or more reviews may be held.

**Software Test Plan Review** - is held to ensure that every requirement in the Software Requirements Description will be tested adequately.

**Software Project Plan Review** - is held to verify that the Project Plan is adequate and reasonable.

**Software Configuration Management Review** - is held to verify that project specific CM is adequate and reasonable.

**Code Reviews** - are held to detect defects which often cannot be found by testing and to verify compliance with the project coding standards. Code reviews range from off-line peer reviews to formal inspections.

## 6.5 Sandia Required Software Activities

YES: this element must be included
REC: This should only be excluded for a good reason. It is recommended
OPT: Including this element may be appropriate for this project, but it is optional
NO: This element does not have to be included

**Table 2: Sandia Project Documentation**

| | I | II | III | IV |
|---|---|---|---|---|
| Software Requirements Spec | YES | YES | REC | OPT |
| Operator Interface Spec | REC | OPT | OPT | NO |
| Software Design Description | YES | YES | REC | NO |
| Software Test Plan | YES | REC | REC | NO |
| Software Test Results Summary | YES | REC | REC | NO |
| User Documentation | YES | REC | OPT | NO |
| Software Project Plan | REC | REC | REC | NO |
| Software Config. Mgmt Plan | YES | REC | NO | NO |

**Table 3: Sandia Project Reviews**

| | I | II | III | IV |
|---|---|---|---|---|
| Software Requirements Spec. Internal Review | YES | YES | REC | NO |
| Software Requirements Spec. External Review | YES | REC | REC | NO |
| Operator Interface External Review | REC | OPT | OPT | NO |
| Operator Interface Internal Review | REC | OPT | OPT | NO |
| Strategic Design Review | YES | YES | YES | NO |
| Software Design Review | YES | REC | REC | NO |
| Software Test Plan Review | YES | OPT | OPT | NO |
| Software Project Plan Review | REC | REC | REC | NO |
| Configuration Mgmt Plan Review | REC | OPT | NO | NO |
| Code Reviews | REC | OPT | OPT | NO |

## 7.0 Developing an ISD Process Model

To effectively propose a process model for ISD software development to follow, several factors should be considered. Certainly, the conclusions discussed in the Findings section are important. A balance must be achieved among honoring the flexibility that often feeds creativity, the current culture within ISD, and the perceived benefits from a more structured process. The software engineering world has become process-oriented, giving the false impression that instituting a software process is a panacea that cures all ills. However, processes do not create software, people do. This is not to say that anarchy is the way to success. The proposal emphasizes an evolutionary and experimental approach to processes. It is helpful to think about a process as a pattern for solving a problem.

An initial model is suggested. Implementation of the model is not the end of the experiment. The results must be tracked along the way, and adjustments made. This is an adaptive process. The meta-model is one of "continuous improvement," where an experiment is tried and lessons are learned. If an approach does not work, we figure out what to change so that it does work. A pilot project should be selected to try some or all of the elements of our proposal. Members of the Software Process Team would work with the project team and gather results as the project progresses.

## 7.1 Categories of Software Efforts

Within ISD there is wide variance in the types of software efforts undertaken, which is similar to the Intelligent Systems and Robotics Center (ISRC) at Sandia. Variables influencing a software project characterization include:

**Category of deliverable:** demo only, executable released to customer, source code released to customer, utility for use internally or externally

**Consequence of Software Failure:** are humans endangered by software failure, or would valuable equipment be potentially ruined?

**Complexity of project**: scope of undertaking, number of developers collaborating

**Longevity of project**: will the project be used only as a proof-of-concept, or is it the foundation for additional work over the next few years?

**Use of RCS architecture**: because this is a major thrust, and we seek to develop a methodology for design of control systems using RCS, special attention should be paid to projects which are either built using RCS technology or seek to further the RCS framework.

## 7.2 Proposed Project Categories

Believing that it would be counterproductive to have too many gradations of project categories, we propose a three-tiered system. The criteria for discrimination among the categories are:

I. System or code will be released outside ISD; or the code is a utility which will be used internally by potentially several projects; or the system requires high reliability due to potential danger to humans or equipment.

II. The system is not delivered outside ISD and is not meant to be a utility. However, either the number of developers needing coordination, the projects's future potential, or the use of RCS architecture warrant a higher level of rigor.

III. The system will not be delivered outside of ISD, is not meant to be a utility, and its longevity is limited.

## 7.3 Elements of the Process Model within ISD

**Project Design Review** - Division management, project leaders, and selected technical personnel attend this review. The scope, requirements, and technical approach for the overall project are presented. The project review will cover the entire project's scope, which typically involves more than purely software. During the project design review, the necessary software development tasks should be outlined. At this time, it may be appropriate to emphasize the need for prototypes or experiments. The risk factors should be highlighted to guide such decisions.

**Software Design Document(s)**

**Software Design Review(s)** - The main components of the software system are described: the requirements should be explained, the trade-offs that are being considered, the issues that are still open. Areas, such as performance requirements and how they will be addressed, should be covered where appropriate. It is important to have all the project team members and other technical staff who can evaluate the design and look for areas of reuse attend. If a particular subsystem seems to be a candidate for becoming a utility, this can be noted and explored after the review. Several design reviews may be necessary to cover a particularly complex or large system.

For RCS-based systems, a high-level review should include an operational scenario, along with the proposed control hierarchy.

***Issue***: What language do we use to communicate software designs? A common language and common tools across the division are highly desirable.

**Operator Interface Design Document**

**Operator Interface Design Review** - Where applicable, the Operator Interface for a system is described in the Design Document. Mock-ups of screens, forms, and operational scenarios should be included. The underlying design to achieve the desired operator interface is also presented. If ISD begins to use a GUI toolkit, for example, certain forms or lower-level constructs should be reusable. This type of reuse should be sought in the review. If appropriate, a separate audience can review the pure operator interface. For example, the recipients of the system may not care about the implementation, but are very concerned with the mechanics of how to interact with the system.

**User Documentation**

**User Documentation Review** - When an API or code is delivered outside NIST, user documentation is appropriate. If some or all of the code within a project will become a utility within the division, user documentation is also required.

**Quality Plan**

**Quality Plan Review** - If code is going to become part of the utility set or if it is going to be used outside of ISD, reliability needs to be assured. The quality plan should describe the steps to be taken to ensure that the code meets the requirements - in terms of functionality, performance, and reliability. Use of tools such as Centerline, Purify, and PureCoverage are encouraged.

**Table 4: Requirements by Category**

| | I | II | III |
|---|---|---|---|
| Software Design Document(s) | YES | YES | OPT |
| Software Design Review(s) | YES | REC | OPT |
| Quality Plan Document | YES | REC | OPT |
| Quality Plan Review | REC | OPT | OPT |
| Operator Interface Document | YES[a] | REC[a] | NO |
| Operator Interface Review | YES[a] | OPT | NO |
| User Documentation | YES | OPT | NO |
| User Documentation Review | REC | NO | NO |

a. If applicable

| | |
|---|---|
| YES - Required | NO - Not necessary |
| REC - Recommended | OPT - Optional |

REC and OPT are left to the Project Leader’s and Division Management’s discretion.

## 7.4 Relationship of Process Model to ISD Project Management

ISD has been developing a general project management model which is to be followed by all division projects. Project leaders should generate plans that indicate the main tasks, milestones, resources, and time frames. Projects which are to follow the ISD Software Process model should explicitly include elements of the process model in their plans and schedules. In Figure 3, a project’s major phases are shown as a vertical line on the right. The phases are the fairly standard: design, code, unit test, integrate, final test. In this example, elements of the proposed software process are inserted as milestones within the project’s plan. The project plan review is shown as occurring in the early design phase. Several software design reviews may occur. The initial project review is shown to encompass an early software design review. At the end of the design phase, another, more comprehensive and detailed software design review occurs. In the same manner, quality plan, user documentation, and operator interface reviews are inserted into the project’s schedule where appropriate. The necessary documents to support the design reviews are also shown as being part of the “deliverables” for the milestones. This example shows a very rigorous set of requirements for the project merely to illustrate the concept.

More realistically, at the initial project review, the project will be placed within one of the three categories shown in Table 4. The requirements for deliverables and reviews would be determined at that time. At this early point in the project’s life, assessments of potential components which can be reused should also be discussed so they can be reflected in the design of the system.

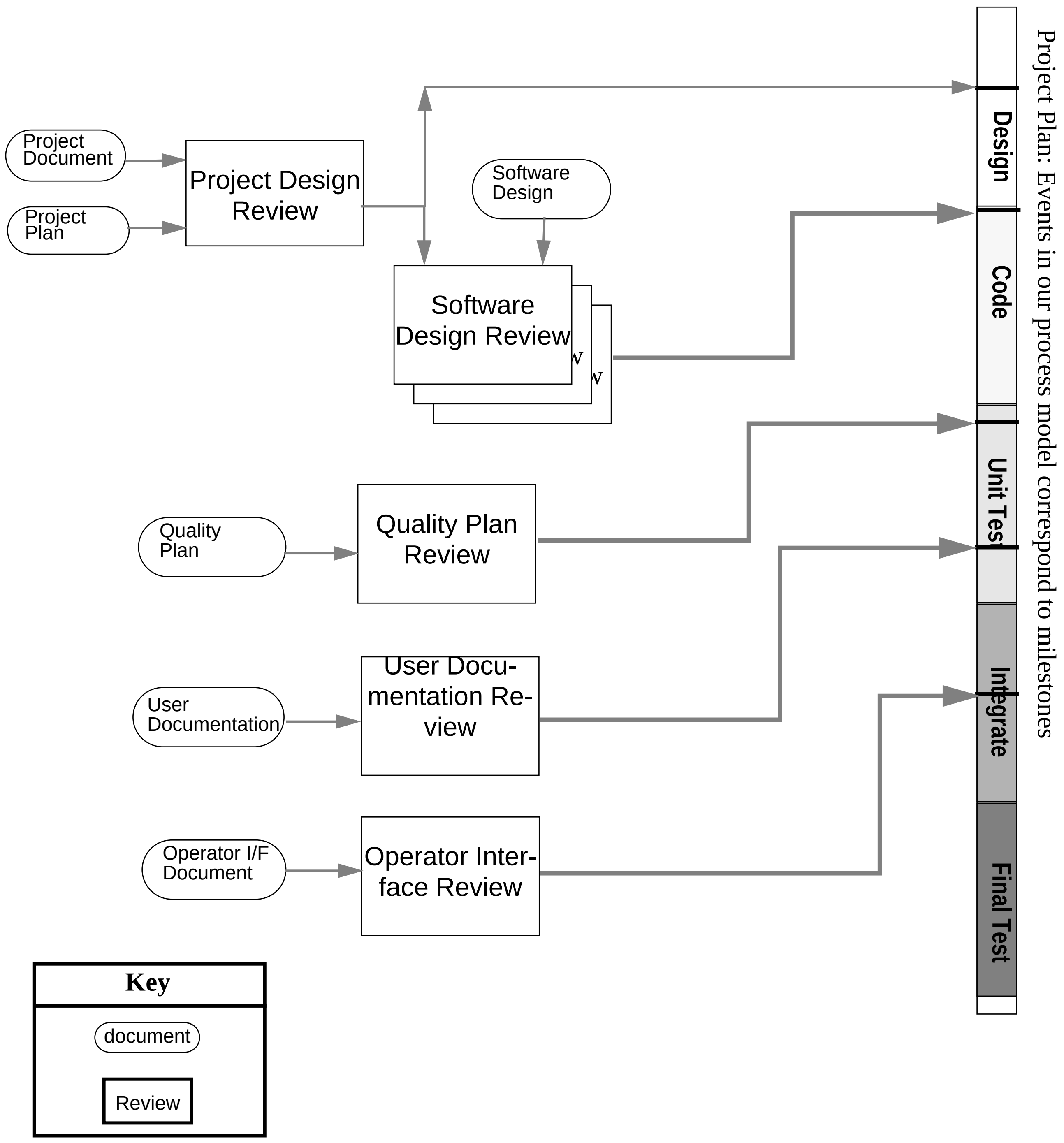

**Figure 3: Example of Process Elements and Their Relationships to Project Plan**
Successful Completion of a Review is an Explicit Milestone in Schedule

## 8.0 Overall Recommendations for Short-Term Actions

Based on the above responses to perceived areas needing attention, we now try to unify the concepts into a short-term plan of action. This effort will not succeed if it is viewed as *the* solution to all our software development issues. The primary recommendation is to assume an experimental approach to the proposed actions. Feedback throughout the process is essential to correct the course. The attitude of the division leaders and participants should encourage and anticipate revisions to the proposed models.

A project should be selected in the near-term to serve as a pilot. Ideally, it would fall into project category I or II, be RCS-based, and involve at least two developers. The potential project team members and project manager would meet with the Software Process team to understand the goals and discuss the pilot program. If everyone agrees, we would proceed. The project would be classified per our project categories. The project manager would agree to adhere to the required document deliverables and reviews. A milestone chart would be devised and maintained. According to the evolutionary model of software development, the longer people work on a project, the smarter they are about it, and the milestones and schedules are refined. Tentative formats for the documents and review materials will be developed. It is recommended that a tool, such as Rational Rose, ControlShell, be selected for design and documentation. We probably can have the Software Process project underwrite the cost of tools purchased. The development directory structures and configuration management will be implemented to better address the project needs.

For each of the action items and experimental approaches associated with the pilot project, the Software Process team and the project team should receive feedback. Lessons learned will be incorporated into the Software Development Process document (a follow-up to this).

The pilot project will incur additional costs above and beyond its initially planned budget (pre-pilot). The project manager and project engineers will have to spend additional time on the following activities:

- meetings and briefings with the Software Process team
- becoming familiar with general concepts which will be used
- selection of a tool
- learning to use the tool
- producing the various documents as required by the process
- participating in the various reviews as required by the process
- if reuse becomes a major priority, it will probably require more time to implement the initial system.

The project may indeed finish close to initially estimated time. However, note that, typically, most projects do not complete close to the initially estimated time. Therefore, it is possible that the process will pay off in terms of more time spent on design and review, and less time on integration and debugging. Longer-range, the benefits should accrue. If certain portions of the pilot project can be reused, benefits to the division begin to accumulate. If the selected tool proves to be useful, the pilot team will be able to continue using the tool on other projects. In this way, the team can serve as an educational resource for others.

Recognizing that any project team that agrees to serve as a “guinea pig” for the software process will accept responsibilities above and beyond their project’s challenges, we feel that it may be important to provide incentives to the pilot project. For example, the Software Process project could underwrite the cost of the chosen tool. Depending on the funding scenario, the Software Process Project could also subsidize the pilot project to account for the extra responsibilities.

## 9.0 Conclusions

After examining the various facets of software development within the Intelligent Systems Division, the Software Development Process Team has produced a summary set of conclusions and recommendations. The set of recommendations seeks to balance the research nature of ISD’s work with the desire for higher software productivity and quality. A set of project classifications with corresponding process models has been set forth. Suggested actions to address specific issues within the development organization have also been outlined. We believe that the ultimate solution will be to embark on a controlled experiment with a pilot program. Only when these concepts are tested, will we begin to take steps towards improving the software development process in ISD.

## 10.0 References

[1] Naur, P., and B. Randell (eds.), Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, NATO, 1969.

[2] Pressman, Roger, Software Engineering: A Practitioner’s Approach, McGraw-Hill, Inc., 1992.

[3] Humphrey, Watts S., Managing the Software Process, Addison-Wesley, 1990.

[4] Boehm, B., “A Spiral Model for Software Development and Enhancement,” Computer, Vol. 21, No. 5, May 1988, pp. 61-72.

[5] Wise, Cindi, “A Study: Senior Management Actions Critical for Successful Software Process Improvement,” Process Inc. Newsheet, Vol. 1, No. 3, December, 1994.

[6] “Preferred Processes for Software Development”, Sandia National Labs Internal Publication, 1994.

# Appendix A

# A Vision of ISD Software Development in 1998

A new research project is underway. Several developers are selected to begin working on it. A hardware and operating system platform must be chosen, and several other requirements need to be determined. An Other Agency (OA) sponsor wants to see an RCS implementation on a parallel processing machine with distributed nodes and a minimum of three sensors. The demonstration must be coordinated with two other parties, Egregious Enterprises (EE) and Zap Software (ZS). There is already a target schedule for the initial demo. What happens next?

The project leader and team members meet with the sponsor and the partners (EE and ZS). Requirements (or constraints) are determined: software and hardware choices, based on desired sensors and decision to support distributed and parallel hardware; minimum and maximum performance targets, interfaces between major subsystems (especially to those provided by EE and ZS), procedures for integration and testing, configuration management, and first-pass schedule for first iteration. A spiral model is assumed.

Hardware and software choices are made. The Systems team participates in this choice, providing background information on supported systems. Given their experience and convincing arguments, a choice which will leverage an already **sanctioned platform** is made. The systems team will have to expand the communications utilities due to the distributed and parallel nature of the effort. At this point, a **project review** takes place with the division management. The direction and strategies are evaluated and approved by the management review team. This project evidently is following technological development and implementation strategies which are consonant with NIST, MEL, and ISD goals. Given the green light after the project review, a designated senior member of the team produces a high-level design specification for the system and presents it in a **design review** to the review team. Based on their expertise in the given domain, the design reviewers are selected from across the division. A representative from the Systems Team is included, as usual. This high-level design is represented using the appropriate methodology and tools, with which most people in the division are familiar. The design review team provides input on which existing algorithms, code, or libraries should be **reused** by this project. The applications project team produces a more detailed level design of their system.

RCShell,[1] a control system design tool with a graphical front-end, is used for the design and analysis. Designers follow the RCS doctrine and build a system which has the appropriate number of hierarchical levels. RCShell provides guidance in these design issues by asking questions which suggest the correct operational parameters. Libraries of components are available to the system designers: servo, sensor, matrix operations, even operator interface components can be linked together graphically via RCShell. State tables, of course, are easily generated by RCShell. RCShell provides facilities for defining the knowledge base / world model for the system (which can be distributed). Communications protocols for passing commands and information can be specified via RCShell. The inputs and outputs to the components provided by EE and ZS have an initial definition so they can be modeled in RCShell. Because some new sensors need to be added to the system, new components for them are designed and added to the RCShell library. The pro-

1.pronounced “RCS Shell”

cedure for adding a new component is fairly straightforward.

Once the higher level design is completed, showing the hierarchy, major components, data flow, and projected timings, a **design review** is held. The designated reviewers receive pointers to review materials a couple of days beforehand, allowing them to **run the review prototype** ahead of time. Thanks to RCShell, an annotated simulation of the system can be accessed in the **project's home page** through the ISD Web. This is very convenient for the California sponsor, as well as the other development sites. The review material includes the usual information, such as which components are expected to be "off-the-shelf" versus which ones need to be developed, allowing for refinement of the resource estimates and schedules. Open issues and decisions to be made are also included in the review information. One of the major open issues is the need to experiment with the processor configurations to see if the desired performance can be achieved without incurring an excessive inter-processor overhead. The project home page contains the project schedules, which are kept current on a regular basis. The ISD management can track the project's progress via the home page likewise.

The review prototype also includes a **prototype operator interface.** Using the RCShell, one of the developers put together the various screens for the operator interaction. This allows the various team members to look at the OI and even take it for a test run, which qualifies as an **Operator Interface Design Review**.

The design review results in some modifications being required. The applications design engineer responsible for these changes goes back to RCShell and accommodates the modifications. An updated design and simulation is released, to be reviewed on-line by the same review team. The changes seem acceptable to all. The schedule was modified accordingly, because an additional component will be needed. The initial design review also points out the need to hold another review over the algorithms that handle the sensor fusion.

The sensor fusion algorithms are developed using RCShell, which is integrated with an existing mathematical modeling package. RCShell's **integrated documentation facility** is a real boon at this point. The updates to the design in RCShell are annotated with change history and the tool facilitates generating module documentation. A design review is set up on the WWW, once again allowing geographically-dispersed reviewers early access to the graphical, textual, and simulated components of this subsystem's design.

- interfaces between pieces
- generating templates; defining diagnostics desired
- adding components to RCShell vs. using existing ones
- validation of new generic components added
- associativity between documentation and code
- configuration management (subsystem releases; project configuration setup in the 1st place; notification of team members of changes to documentation/code

Note that the above is not meant to detail a completely realistic scenario, especially technically, Rather, it is meant to provide an impressionistic view of a possible way to work at a future date.