# Process Enactment Within An Environment*

Roseanne Tesoriero and Marvin Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland

February 29, 1996

Abstract

Environment research has often centered on either the set of tools needed to support software development or on the set of process steps followed by personnel on a project as they complete their activities. In this paper, we address the effects that the environment has on the development process in order to complete a project. In particular, we are interested in how software process steps are actually performed using a typical programming environment. We then introduce a model to measure a software engineering process in order to be able to determine the relative tradeoffs among manual process steps and automated environmental tools. Understanding process complexity is a potential result of this model. Data from the Flight Dynamics Division at NASA Goddard Space Flight Center is used to understand these issues.

## 1 Introduction

Since 1976 the Software engineering Laboratory[1] (SEL) at the NASA Goddard Space Flight Center (GSFC) has been studying flight dynamics software development and has produced over 300 papers and reports describing numerous process improvement technologies. Figure 1 briefly summarizes this activity and outlines the SEL approach to understand, assess, and package technologies useful for the flight dynamics domain. Many of these technologies (e.g., resource models, cleanroom, IV&V, object-oriented design) have been studied and some made part of the SEL development process.

There are two interesting issues not answerable by the set of studies addressed by this figure:

1. Software productivity has obviously improved in the 20 years since the SEL started (and 20 years worth of workshop proceedings attest to that). But by how much? Also, *everyone's* productivity has improved in these past 20 years. Is the SEL doing relatively better than other development organizations? How do we understand this and measure this improvement?
2. What impact has technology (i.e., tools) had on this improvement? Most SEL studies are on *processes* that personnel undertake in the develoment of software. What impact has the set of computers, workstations, and software had on development productivity and quality?

*Research supported in part by NASA grant NSG 5123 to the University of Maryland.
[1]A joint activity of the NASA/GSFC Flight Dynamics Division, Computer Sciences Corporation (CSC) and the University of Maryland.

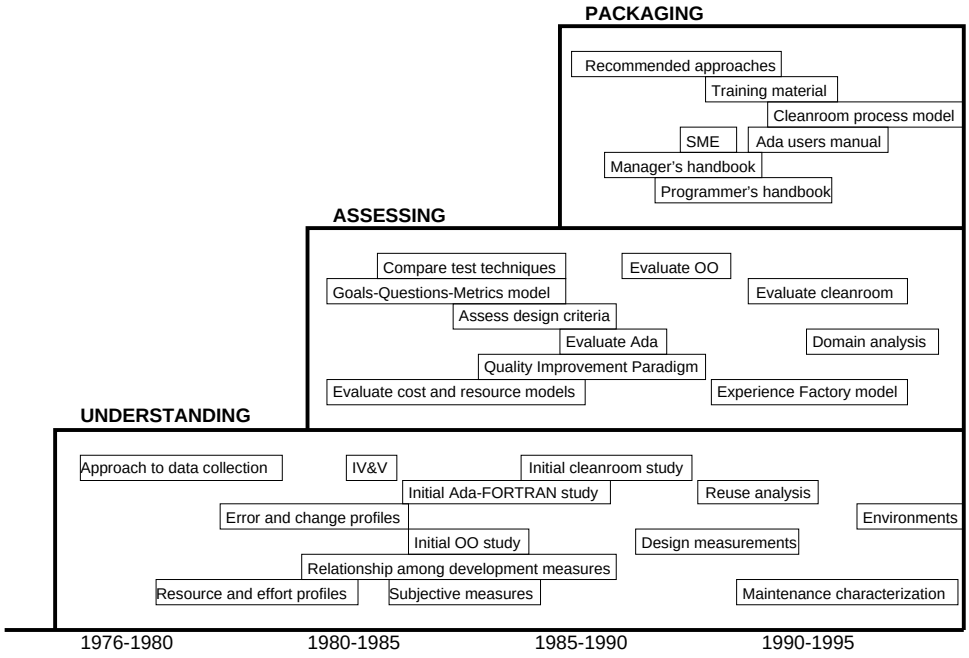


Figure 1: Summary of SEL activities.

The first issue can be explained by the following analogy. Assume an organization purchased four PCs over the past 15 years:

- 8088 4.77Mhz PC-XT with 10 Mbyte disk bought in 1983 for $5,300
- 80286 6Mhz PC-AT with 20 Mbyte disk bought in 1984 for $4,000
- I486Dx2 66Mhz PC with 340 Mbyte disk bought in 1993 for $2,600
- Pentium 75Mhz PC with 850 Mbyte disk bought in 1995 for $1,700

The fourth system (the Pentium) is obviously the fastest most powerful system of the four. However, the more interesting question is which is most powerful relative to the era in which they were available? Is this organization doing better or worse than other organizations in its industry relative to computer power if it followed this purchase plan?

For answering this question, we have many hardware performance measures: LINPACK, TPC-A, $/mips, dhrystones, etc. All have their faults, but at least they generate a set of numbers that can be discussed. How would we do the same for software development practices? We have no such measures of performance. It is toward this end we are trying to develop a model.

This then focuses the remainder of this paper. We are interested in a process measure that addresses the following issues:

1. *How do we model such a process performance measure?* We will base our model on a service-based model of an environment.
2. *What is an environmental service?* We will describe such a model of an environment service.
3. *What tools are used in the SEL environment?* We looked at the evolution of tool use within the SEL to see how tool use reflects the set of services in our environment model.

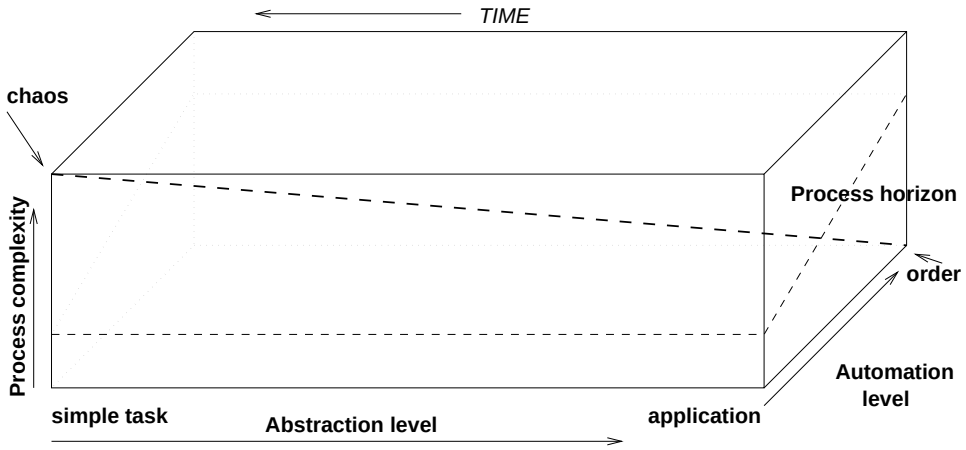


Figure 2: ITEM Model.

4. *How do we classify flight dynamics processes?* Can we adapt this service-based model to the flight dynamics application domain better? Can we measure the complexity of a process according to this model?

The next four sections of this paper, in turn, addresses each of these questions.

## 2 How do we model such a process performance measure?

The model we choose to use is an extension to a model developed by Zelkowitz and Cuthill at the National Institute of Standards and Technology. Called the Information Technology Engineering and Measurement (ITEM) model [16], it addresses the role of both automation and process in an environment.

The model (Figure 2) locates a given development activity according to three parameters:

1. Percent automation (ranging from a manual process to totally automated one).
2. Service abstraction level (from simple low-level infrastructure to an entire application domain).
3. Process complexity

We are interested in understanding activities from these three perspectives. We want to know how much of a solution the activity addresses. A tool that does compilation is relatively simple, while one that integrates design, compilation, and testing (assuming such a tool existed) would be more complex, and finally one that by pushing a button configures an entire ground support system for NASA would be even more complex. We call this aspect of an environment the *abstraction level* and is represented by its horizontal placement in the figure.

For any specific activity, there are multiple ways to address its solution. Take for example, sorting a list of names. A totally manual process would be to sort each name by hand from a set of cards, each containing one name. A more automated solution would be the UNIX *sort* command on a file of names. This is the *automation level* of that activity.

Finally, for each activity, there is a limit to the complexity that can be attempted in its solution. For example, sorting a list of names where one first translates by hand each name into binary, sorts that binary list by hand, and then translates the binary back into ASCII, probably represents a solution that is fraught with errors. We call such a solution too complex, or above the *process horizon* for that activity. The process horizon is represented by the dotted line in Figure 2. Note that the line rises as the automation level increases – more automated tasks can handle more complex solutions due to the increased use of computer-based tools that handle the mundane part of the solution.

The point on the model labeled "chaos" represents an extremely low-level manual task of great complexity. This represents a chaotic state of development. On the other hand, the point labeled "order" represents a totally automated solution of the application domain with simple complexity. This is our desired goal. So our problem is to try and describe the location of each process in this ITEM structure and then to show that over time we progress toward the point labeled "order."

*Time* is a parameter that makes the application of this model even more complex. As technology improves over time, complex tasks of last year are now relatively simple. A complex task like assembly code in 1955 was replaced by the similarly complex task of compiling a Pascal program in 1970, which was replaced by the more similarly complex task of writing a C++ or Ada program today. We can quibble with the term "similiarly complex," but the basic concept is that all of these tasks represented about the same level of effort in their respective eras. However, today, the same assembly language compiler would be considered a more complex lower-level activity than the Ada compiler, and thus would be to the left and higher (more process complexity) in the ITEM figure. This concept is called *technological drift*. All technology moves to the left over time in the ITEM model. A future goal (one we would like to solve, but dare not even try yet) is to measure this flow of technology over time.

# 3 What is an environmental service?

The ITEM model describes an abstraction level as a way to characterize the functionality of a given activity. For the software development application domain, two service-based models have been developed that address this functional – the NIST/ECMA framework for software engineering environments and the Project Support Environment (PSE) reference model.

## 3.1 Reference Models

In 1989, the European Computer Manufacturers Association (ECMA) produced a model for the description of services useful for supporting software development activities. In 1991, the National Institute of Standards and Technology (NIST) joined with ECMA to develop the current Edition 3 of this model, known as the NIST/ECMA software engineering environment frameworks reference model [11].

The NIST/ECMA model, also known as the "toaster model" based on a graphic that was developed by George Tatge of HP (Figure 3), defines the underlying infrastructure set of services for supporting tools executing on a software engineering environment. The model consists of 66 services catalogued according to the classification of Figure 3 plus a seventh Operating System set of services that supports the other six categories. Software products, called tools, are added to the environment and interact among one another and with the environment itself by using the operations defined within the seven classes of infrastructure services.

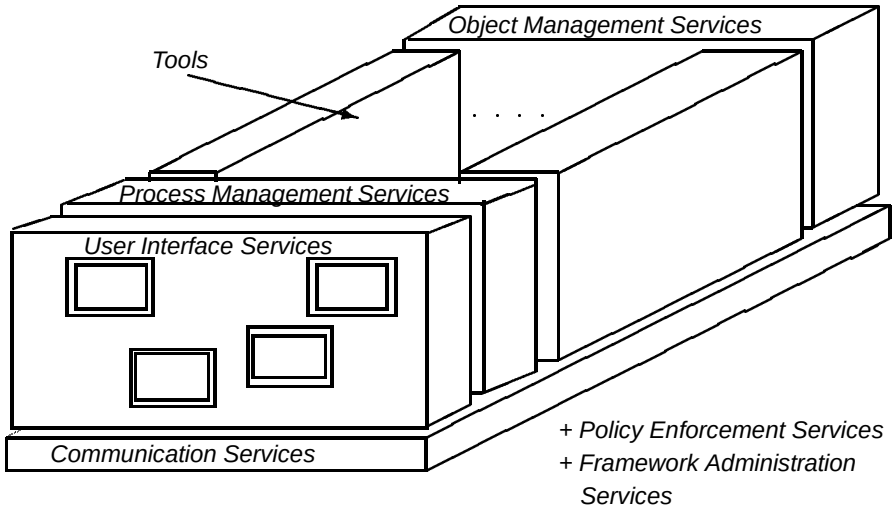


Figure 3: Framework Reference Model Service Groupings.

While the NIST/ECMA model defines the underlying infrastructure for supporting tools within an environment, there was also a need to define the set of tools that users would need for application development within the environment. In order to address this functionality, the U.S. Navy's Next Generation Computer Resources (NGCR) program created the Project Support Environment Standards Working Group (PSESWG). PSESWG developed the Project Support Environment (PSE) Reference Model of the set of services needed to support users of software engineering environments [5]. This model included the NIST/ECMA framework model as the core set of framework services, but adds a structure on the "tool slots" of the framework model.

Services in the PSE model are either *end-user* or *framework*. The former services directly support the execution of a project (i.e., services that tend to be used by those who directly participate in the execution of a project such as programmers, managers, and secretaries). The latter services generally pertain to the operation of the computer system itself (e.g., a human user performing such activities as tool installation) or are used directly by other services in the environment. End-user services are further subdivided into Technical Engineering, Technical Management, Project Management, and Support service categories.

For example, within the Software Engineering group of services within the Technical Engineering category are the common software development activities, such as software design, Compilation, Debugging, etc. Project management services include project scheduling, estimation, and tracking. Support services include activities such as editing, publishing, electronic mail, and other supporting activities.

The model has been used to map (i.e., describe and contrast) the functionality of various products or standards in order to determine how the functionality they provide compares to the functionality present in the model [14].

## 3.2 Service Hierarchy

**Process Steps:** We will define a process step as the smallest identified self-contained activity performed in the development of a software project. Personnel on a project perform process steps, and the underlying computer environment executes one or more *tasks* in response to those steps. For many process steps, there is a one-to-one correspondence between process steps and tasks (e.g., editing a file and a tool such

| Process Step | Sample tasks (tools) |
|---|---|
| Compilation | Ada, C, or FORTRAN compilers, preprocessors |
| Debugging | symbolic debuggers |
| Software Build | make |
| Text Processing | editors such as VI or EMACS |

Figure 4: Level 1 Services.

as VI or EMACS, placing a file under configuration management and tools such as PANVALET or SCCS). In other instances the process step is clearly defined by a small set of tasks (e.g., creating a software module requires the tasks of editing and compiling). However, other process steps may require manual components as part of the process step, which may not involve the use of a computer (and its execution of tasks) (e.g., software design, inspections, software quality assurance). In these cases, process steps may be implemented by a sequence of actions using process execution notation, such as MARVEL [10].

**Reference model services:** The tasks provided by an environment can be mapped into the end-user services defined in the reference model. These may also be viewed as an initial set of process steps useful in the software development process.

The services in the reference model represent a broad range of capabilities. Some are clearly crucial to the success of a project (e.g., the existence of a compiler for the compilation process step). Without these services, the development of software would be impossible.

Other services, while not necessities, help to improve the quality and reliability of a software system. For example, program verification helps to improve the correctness and reliability of a program. However, the difficulty of using verification tools and the general lack of availability of effective verification tools means that most development processes omit a verification step.

One way of representing the relative complexity of the services is to impose a hierarchy on them. The levels in the hierarchy indicate increasing levels of complexity in the services provided by the project support environment. Because of advancements in technology and research, we believe that the levels will continuously evolve and change. Currently, the services of the lower levels of the hierarchy represent steps in the software process that are well defined and understood. We can easily identify tools that provide these services. As the levels of the hierarchy increase, the current understanding of the process steps required to provide the services of the level decreases. It becomes increasingly difficult to find examples of tools which provide these services. We believe that over time, as the knowledge and understanding of complex services become clearer, the services will drop to the lower levels of the service hierarchy.

We propose the following four complexity levels of the reference model end-user services:

**Level 1: Basic Services.** These represent the crucial processes in software development. Software development would be impossible without them, and all are present in every realistic development. In addition, they are all implemented as single tasks in an environment. The process steps and example tools for these services are given in Figure 4.

**Level 2: Single Task Services.** These services represent the process steps that are usually performed in an environment. The process steps often exist as single tasks in an environment and tools to automate

| Process Step | Sample tasks (tools) |
|---|---|
| Configuration Mgmt. | PANVALET, CMS, SCCS |
| Software Generation* | LEX, YACC, 4GLs, IDL tools |
| Software Testing | Software TestWorks |
| Software Static Analysis* | source code analyzers, SAP |
| Numeric Processing | spreadsheets such as QuattroPro and 1-2-3 |
| Figure Processing | MacDraw, MacPaint, xfig |
| Mail | ccMail, Eudora |
| Bulletin Board* | Readnews, ccMail |
| Tool Installation and Customization | UNIX .rc tool customization files |
| Host-Target Connection | Xterm, ftp, telnet |
| Audio and Video Processing* | MBone |
| Calendar and Reminder* | Synchronize, CRON |
| Conferencing* | Unix talk |
| Information Mgmt.* | Archie, Gopher, yellow pages, parts lists, Xmosaic, Netscape |
| Planning | TimeLine, MacProject |
| Estimation | Software Management Environment (SME), tools that implement cost models |
| Tracking | SME, Gantt and Pert charts) |
| Data Interchange* | binhex, uuencode/uudecode, Kermit |
| Publishing | Tex, Latex, Framemaker, PowerPoint, MS Word |

* optional process steps

Figure 5: Level 2 Services.

them are readily available. Most of the listed services are typically utilized in an environment; however, some, although automated and readily available, are not necessarily required. Those services that are considered to be optional are indicated by an * in Figure 5.

**Level 3: Composite Services.** The services at this level represent process steps that are required in software development, but there is rarely a single tool which can be used to accomplish all of the activities of the process step. These process steps generally represent the state of the art in software engineering research. As the table in Figure 6 demonstrates, there are not as many tools available for for implementing these as single tasks within an environment. As technology evolves and improves, it is expected that these services will become single task services.

**Level 4: Advanced Services.** The services in this level represent needed research in software engineering. Not every project incorporates these process steps into their development process. Often, if tools are available for these services, they are experimental. It is expected that as more research and experimentation is completed, these services will move to Level 3 services. The process steps and examples for these services are given in Figure 7.

| Process Step | Sample tasks (tools) |
|---|---|
| System Integration | |
| System Testing | |
| System Static Analysis | |
| Software Requirements Engineering | OOATool and DCDS |
| Software Design | IDE's Software through Pictures (StP), Teamwork ObjectMaker |
| Software Simulation and Modeling | Simula, screen simulators |
| Software Traceability | ORCA (Object-based Requirements Capture and Analysis) and RETRAC (REquirements TRACeability) |
| Change Management | Netherworld, ChangeVision |
| Reuse Management | Asset Source for Software Engineering Technology (ASSET), Central Archive for Reusable Defense Software (CARDS) |
| Metrics | Amadeus, COCOMO (COst COntainment MOdel), cyclomatic complexity structure models |
| Annotation | |
| Process Management | MARVEL, shell scripts |
| Target Monitoring | |

Figure 6: Level 3 Services.

| Process Step | Sample tasks (tools) |
|---|---|
| System Requirements Engineering | |
| System Design and Allocation | configuration languages and systems (Polylith, Rapide) |
| System Simulation and Modeling | Performance Oriented Design (POD) and Synthetic Environments for Requirements and Concepts Evaluation and Synthesis (SERCES) |
| System Traceability | ORCA (Object-based Requirements Capture and Analysis), RETRAC (REquirements TRACeability) |
| Software Reverse Engineering | |
| Software Re-engineering | |
| Process Definition | state transition charts, MARVEL, action diagrams, Petri nets |
| Process Library | Process Asset Library (PAL) |
| Process Exchange | |
| Process Usage | EAST, Cohesion |
| Risk Analysis | |
| System Re-engineering | |
| Software Verification | Cleanroom, Z, VDM |
| Policy Enforcement | |

Figure 7: Level 4 Services.

# 4 What tools are used in the SEL environment?

In order to study tool use and process interaction, we are using data collected by the NASA/GSFC SEL. The SEL has been collecting data since 1976 on over 125 ground support software projects for unmanned spacecraft developed by the Flight Dynamics Division. The data in this paper is a result of reading history reports on lessons learned from approximately 20 such recent projects, and from interviews with personnel who built this software.

## 4.1 NASA Flight Dynamics Software Development

Two classes of products form the core of the work within this division:

(1) Attitude Ground Support Software (AGSS) provides attitude determination capabilities for determining spacecraft location and orientation (e.g., where is it and in which direction is it pointing at a given time in the future). This is needed to coordinate spacecraft location with the data collected by on-board scientific instruments and for spacecraft orbit modification.

(2) Simulators for testing onboard computer capabilities before launch.

Until the late 1980s, all source programs were written in FORTRAN. AGSS software is still written in FORTRAN, while most simulators are now written in Ada. Other software is written in FORTRAN, in Ada, or in C. There is now a project investigating the applicability of C++ in this environment.

The programming (e.g., hardware and software) environment at NASA has generally passed through three distinct phases since the SEL was created in 1976. Initially, all work was performed on IBM-compatible mainframes. In the early 1980s, the DEC VAX computer was used for some development, while the mainframes were still the operational computers for spacecraft control. In the late 1980s, PCs started to appear on desktops and were sometimes used for initial editing and compilation of modules. Today there are some initial moves to use UNIX workstations and build systems using a client-server architecture. The following briefly describes this evolution.

#### IBM Mainframe Environment

The mainframe environment is the oldest of the three development environments. It is also the least sophisticated. The mainframe environment is used for the development of AGSS systems with FORTRAN being the programming language used for development. In order to integrate the various tools available in the IBM mainframe environment, the Software Development Environment (SDE) was built. SDE is an application that was built on top of ISPF (Interactive System Productivity Facility). It is a menu-driven facility that provides a common interface to the tools available in this environment (Table 1).

**Design:** In the mainframe environment, there is limited automated support for software design. On three projects (GRO, UARS and EUVE), DesignAid CASE 2000 was used for design. The history reports indicate that the tool was useful in some areas such as drawing design diagrams and organizing documentation; however, it was not utilized to its fullest potential. One reason cited is that the task of re-entering data flow diagrams and data dictionaries from the functional specifications document to the PC was too time consuming.

The reuse rates for these early FORTRAN projects were low, approximately 10 percent, while later

| Activity | Tool |
|---|---|
| Design | DesignAid CASE 2000, MacDraw, CorelDraw |
| Code | SDE*, ISPF, QED , VS FORTRAN, assembler, linker, RXVP80, ICA, GESS Display Builder* |
| Configuration Mgmt. | PANVALET |
| Test | CAT |

* – Tool developed for use within this environment

Table 1: Tools in the IBM mainframe environment

FORTRAN projects were able to achieve reuse rates between 80 and 90 percent. Part of the increase in reuse rate may be attributed to the creation of two FORTRAN libraries that handle nearly 80 percent of the functionality of new ground support systems. It is interesting to note that the high reuse rates were achieved without the support of specific reuse tools.

**Code:** QED (a line editor) and ISPF (a screen editor) are the editors used in this environment. To create executable code, users use the FORTRAN compiler, the IBM assembler and the IBM linkage editor. Other than editors and compilers, there are few other tools available for use. RXVP80 and ICA are tools for static analysis of FORTRAN code used to detect inconsistencies in program structure or misuse of variables. To generate code for displays, the GESS Display Builder, developed and maintained by NASA over the past 20 years, is used. This tool was to be replaced after the COBE project ended in 1988; however, no replacement has yet been developed.

There is no symbolic debugger available to mainframe users. To compensate for this deficiency, developers typically place debugging statements in their code. On the TONS project, Microsoft FORTRAN and CodeView were used to build code on PCs initially. When the code was considered complete, it was recompiled with the mainframe FORTRAN compiler.

**Configuration Management:** For configuration management, the environment offers PANVALET, a commercial source code management system. PANVALET has been the exclusive source of configuration management in the mainframe environment for over 12 years.

**Test:** There are no standard tools available for software testing and verification. Data for testing are generated with the assistance of FDD developed simulators or by small programs developed by testers. Any software problems that are found while executing the test plan are reported by filling out a Software Trouble Report (STR). On one project, SAMPEX, the STR form was automated. There are no standard tools that support the generation of test plans or the tracking of test cases. The Configuration Analysis Tool (CAT) was used on the ERBS and COBE projects to track discrepancies and test cases. The capabilities of this tool were described as being useful, but limited. Usually, the tracking of the status of software problems is done through spreadsheets on Macintoshes or PCs, as was done on the COBE project.

Although there are no specific tools for test case generation and analysis, the quality of the code has improved over time. The defect rates for early FORTRAN projects were 9.8 development errors per thousand lines of code, while the defect rates for more recent FORTRAN projects are 4.8 development errors per thousand lines of code.

| Activity | Tool |
|---|---|
| Design | System Architect, StP |
| Code | ACS, EDT, LSE, Ada compiler, assembler, linker, debugger, PCA, SCA |
| Configuration Mgmt. | CMS |
| Test | None |

Table 2: Tools in the VAX environment

**VAX Environment**

In 1985 the FDD began to experiment with the use of Ada for software development. It was found that the mainframe Ada was not reliable enough for AGSS development, but the DEC VAX computer provided an environment suitable for simulator development [15]. Although the initial plan was to completely transition from the mainframe environment to an Ada development environment on the VAX, the VAX environment has been used for the development of telemetry and dynamic simulators only. To support Ada development, this environment has the Ada Compilation System (ACS). Users of the ACS have expressed satisfaction with the capabilities provided by this system. Table 2 lists the tools that are available in the VAX environment.

**Design:** There are no standard tools used for the design of projects on the VAX, although several projects have experimented with a few CASE tools. System Architect was used on TONS. The diagrams produced by the tool were considered to be poor in quality. The learning curve and the unavailability of multiple copies of the tool discouraged future use of the tool. The WINDPOLR project experimented with Software through Pictures (StP). The project experienced difficulties with consistency and configuration control when they tried to make changes to the design. The use of StP was discontinued in the design phase.

Initial reuse rates for Ada projects range between 5 and 20 percent. After the FDD became more familiar with the Ada language and began to experiment with object-oriented techniques, the reuse rates increased. More recently, the reuse rates for Ada projects are closer to 90 percent for projects with similar domains. As with mainframe projects, the high reuse rates have been accomplished without the use of specific reuse tools. However, as mentioned previously, higher reuse also occurred in the mainframe FORTRAN environment. This implies that increased reuse is not resulting only from the use of the Ada language; the use of object-oriented technology is having an across-the-board effect on productivity.

**Code:** The ACS provides various services necessary for software development. For editing, a screen editor (EDT) and a language sensitive editor (LSE) are available. To produce executables, the VAX Ada Compiler, VAX assembler, and linker are available. The Performance Code Analyzer (PCA) and the Source Code Analyzer (SCA) provide dynamic and static analysis of code. Configuration management is handled by the Code Management System (CMS). Unlike the mainframe environment, the VAX environment provides a symbolic debugger. Additionally, VAX users can use electronic mail for communication.

For project management, the Software Management Environment (SME) has been used by the WINDPOLR and EUVE projects. SME accesses the SEL database of past project histories and plots the current project's attributes over time (e.g., errors per week, effort per week) as compared to these histories. CSC's Performance Measurement System (PMS) was used to track project resources in terms of schedule, performance and cost by tasks on the GRO and UARS projects. Both of these tools run on PCs.

| Activity | Tool |
|---|---|
| Design | MacDraw, Microsoft Word |
| Code | SCO Unix, HP Desktop |
| Configuration Mgmt. | PVCS, SCCS |
| Test | None |

Table 3: Tools in the workstation environment

**Test:** There are no standard tools used for testing in the VAX environment. The testing process is carried out similar to the way it is done in the mainframe environment.

As with FORTRAN programs, reliability of Ada code has improved over the past 10 years. The defect rates for the early Ada projects were 10.5 development errors per thousand lines of code. More recently, the defect rates are 4.0 development errors per thousand lines of code.

### Workstation Environment

Over the past few years, several projects have been developed on PCs and HP workstations running UNIX. Most new development is expected to transition to this environment over the next few years. There have only been a handful of workstation projects so far: a multi-application user interface system (UIX); multi-application attitude support components (GSS R1); and a mission AGSS application (XTE AGSS). For development on PCs, the Santa Cruz Operation (SCO) Open Desktop environment was used. For the workstations, HP's desktop environment was used. Table 3 lists the tools available in the workstation environment.

**Design:** No design tools have been used on any of the workstation projects. Microsoft's Word was used to draw design diagrams and create the design documents.

The design process for the workstation projects differed from the traditional mainframe and VAX design process. The conventional design process in the FDD calls for all component designs to be completed before the critical design review. For the XTE AGSS project, the first AGSS built in the open systems workstation environment, the design process was modified to include iteration. Each build of the system was designed then coded successively. This led to difficulties in estimating and assessing the progress of the project. Productivity on the initial builds was considerably lower than expected and the size of the applications was underestimated by a factor of three [4].

The difficulty in transitioning to the workstation environment was greater than anticipated. The perspective of the developers had to change from a purely software engineering one to a systems engineering perspective. Because the infrastructures of the mainframe and VAX environments were stable, the FDD developers had little experience with system engineering techniques. This led to unexpected difficulties in design.

**Code:** The desktop environments include tools for editing, compilation and debugging. Because they are UNIX environments, the standard UNIX commands (e.g., diff, grep, ftp, make) are also available. To build X/Motif windows, Builder Xcessory was used on the XTE project. There are several tools that are available in this environment, but, they have not been used successfully. For example, SCCS is available

| Activity | Tool |
|---|---|
| Documentation | Microsoft Word, PC Write |
| Management | SME*, PMS*, BPMS/BEMS*, QuattroPro |

* – Tool developed for use within this environment

Table 4: Tools in the support environment

for source code management, but, for the XTE project, the UNIX file system was used instead. This implies a more manual error-prone process to make sure that only appropriate versions of a system are used at the appropriate time. Similarly, a performance profiler (probe) was not used successfully during the XTE project.

The use of tools such as Builder Xcessory for interface design also indicates a growing trend towards the use of Commercial Off-the-Shelf (COTS) software components in future FDD systems. In the past, because of a stable computing environment, the FDD tended to build customized software development toolsets and supporting infrastructures. When development moved to the workstation environment, the problems addressed by the customized toolsets and infrastructure had to be resolved again.

An Ada development tool set was considered for the workstation environment. Because of the cost, matching the capabilities available on the VAX was not considered as feasible. So, Ada development is still done on the VAX and integrated and tested on the workstations.

**Configuration Management:** SCCS was attempted for source code management on the XTE project, but it was not used successfully. The UNIX file system was used instead. Additionally, PVCS was also used on both the PCs and HP workstations for version control.

**Test:** For testing, the process for workstations is similar to the VAX and mainframe test process. Internally developed simulators generate and send data for testing. The simulators used for the XTE and UIX projects include the TPOCC internal simulator and the GMIS simulator.

### Additional Support

History reports discuss tools specific for the development of flight dynamics software. However, many other technologies are standard in this environment and are not so mentioned. No one today considers devices such as telephones, fax machines, or copies as significant technologies in doing business, yet all are crucial to project success.

The same is true of some computer tools. Electronic mail is all pervasive for communicating among project members, although connections to others via the Internet is not particularly efficient. The NASA/GSFC mail system has undergone several changes over the past few years, and according to the authors of this report, is still substandard with communication to the world outside of GSFC being very slow. ccMail is the current mail system of choice.

In all three of the environments, the activities of documentation and management are supported by the same tools. Table 4 indicates the tools used for these activities.

There were several tools that were used to support the workstation projects that were not UNIX tools.

For instance, MacDraw and CorelDraw were used to create diagrams. These are general purpose graphics programs and not designed specifically for program design applications. General purpose word processors, such as Microsoft Word and PC-Write were used for word processing. QuattroPro is the spreadsheet that was used for tracking status. Additionally, CSC's software estimation spreadsheet, BEMS/BPMS was used. BPMS/BEMS is a spreadsheet tool that has been developed by CSC for project planning and estimation. Depending upon the life cycle model chosen for the project, the tool generates estimated start and completion dates for all phases of the life cycle. The tool is also used for producing charts and reports with the planning and estimation information.

## 4.2 Development Models

In describing the three NASA environments, the discussion centered on the tools that were used (e.g., ISPF, SME, Ada compiler, MS Word) and the processes used to develop modules (e.g., design, testing, documentation). What is the relationship between the processes that need to be accomplished and the set of tools that can be executed to help solve development problems? In this section, we look at the concepts of software processes and programming environments. We describe formal models of each. In the next section, we unify both concepts in order to describe the process architecture of the FDD development environment.

#### Processes

For our purposes, we use the term *process* to mean a set of partially ordered process steps intended to reach a goal [6]. A process is a fairly complex undertaking, such as software design or configuration management. Performing such processes involves the enactment of many of these process steps.

A *process step* is defined as a subprocess of a process or any recursively defined process step. Creating a module or building a version of the system from the configuration management library are generally considered as process steps. An elementary process step is called an *activity*. Activities are composed of *tasks*, the simpliest action under consideration in a development. For the most part, tasks are single executions of a tool in an environment. Compiling a program, editing a file, or checking in a module are considered to be tasks.

There are various approaches to examining process development. One approach is to form a generalized, *ideal* model and refine it to a specific instantiation. The Software Engineering Institute's (SEI) Capability Maturity Model (CMM) for Software [12] and process modeling languages and environments are often used in this manner. Alternatively, a model can evolve and be extracted from data based on previous experiences. This is the approach taken by the Software Engineering Laboratory's Experience Factory. The Experience Factory builds models based on knowledge gained from past software projects and experiments.

**Process-centered Software Engineering Environments.** As described in [7], a process-centered software engineering environment provides assistance to its users by interpreting explicit guidance-oriented or enforcement-oriented software process models. The generic process interpretor, the process engine, is the heart of a process-centered software engineering environment. Guidance-oriented process models give the process performer indirect assistance while enforcement-oriented process models give direct assistance. Often, such environments follow scripts, developed by the process engineer, which define the sequence of

actions to undertake. Such scripts often look like programming language source programs. For example, to create a new module, a simple script could look something like:

```
Loop
    Call Editor (module name)
    Call Compiler (module name)
    If errors, repeat loop
    Call Testing Program
    If errors, repeat loop
End Loop
Check (module name) in Configuration Library
```

It should seem clear that this script will iteratively call the editor as long as either the compiler or testing program finds errors. Programs are only entered in the configuration library if they pass both tests without errors. If the development team is bound by these process rules, complex sets of interactions can be developed for a development team to follow.

Use of such scripting allows for greater control over the development process. For example, one could prohibit checking a module into the configuration library until the testing program succeeded. Data could be collected by automatically calling a data repository program (e.g., Amadeus) at appropriate steps in the script. Cooperative workflow models can be developed as mail is sent from one developer to another informing the new developer of work needing to be done. For example, after checking in a module into the configuration library, an automatic prompt could inform testing personnel that a new module needs to be incorporated into the latest build of the system.

MARVEL [8] [9] [10] is an example of an enforcement-oriented process-centered software engineering environment. The MARVEL environment uses strategies which are predefined by the process engineer. A strategy consists of an objectbase description, tool descriptions, and/or rules that model the software development process. The objectbase description defines the structure (objects) of the project database. The tool descriptions provide the mapping from the objects defined in the objectbase description to the actual location of the tool implementation. The MARVEL rules follow the style of Hoare's assertions for program verification. When the preconditions of an activity are satisfied, it is scheduled for invocation. After the completion of an activity, the postconditions become true and may satisfy the preconditions of another activity. All activities with satisfied preconditions are then scheduled for invocation.

**Capability Maturity Model.** The CMM [12] is a cross between a software quality approach and a process engineering approach. The CMM identifies key practices that state the fundamental policies, procedures and activities for key process areas. These process areas are grouped into five levels of maturity: (1) Initial; (2) Repeatable; (3) Defined; (4) Managed; and (5) Optimizing. The CMM provides a framework for the practices and areas that should be addressed by a process model, although it doesn't specify the process model that should be used. The individual organization utilizes the framework to build a process model that encompasses the practices identified in the CMM. As an organization matures, the process model covers more of the key process areas at higher levels. The CMM gives the organization a strategy for the steps to be taken for continuous process improvement.

**The Software Engineering Laboratory.** The Experience Factory was developed by the NASA/GSFC Software Engineering Laboratory as an empirically-based feedback-oriented model of process improvement.

The SEL collects data both manually and automatically. Manual data includes effort data (e.g., time spent

by programmers on a variety of tasks: design, coding, testing), error data (e.g., errors or changes, and the effort to find, design and make those changes), and subjective and objective facts about projects (e.g., start and end completion dates, goals and attributes of project and whether they were met). Automatically collected data includes computer use, program static analysis, and source line and module counts.

Software development in the SEL is based upon the Experience Factory concept. The Experience Factory [2] [3] is an infrastructure aimed at capitalizing and reusing the life cycle experience and products. The Experience Factory is the basis for the process model driving FDD product development. The Experience Factory is a logical and physical organization with activities independent from the ones of the development organization. The purpose of the development orgainzation is to develop and deliver systems. It provides the Experience Factory with product development and environment characteristics, data and a diversity of models (resources, quality, product, process) currently used by the projects in order to deliver their capabilities. The Experience Factory processes this information and provides direct feedback to each project activity, together with goals and models tailored from previous project increments. It also produces, stores and provides upon request baselines, tools, lessons learned, and data; all presented from a more generalized perspective.

The distinguishing characteristic of the Experience Factory is that the organization defines itself as continuously improving because it learns from its own business, not from an external, *ideal* process model. Process improvements are based on the understanding of the relationship between process and product in the specific organization.

## 4.3 Process Enactment in the FDD

In this section we classify the set of processes undertaken as part of NASA/GSFC FDD software development to understand the impact that the underlying computer system has on this development. We will do this by merging the concepts of processes and environment reference models described in the previous section.

### 4.3.1 FDD Tool Use

From our survey of the three FDD environments, we collected information on various tools used by many projects over the past 10 years (Figure 8). We can classify those tools according to the tasks (i.e., services) that they implement in the PSE model (Figure 9).

The PSE reference model was built around the set of tasks that can be applied to the software development process. For the most part, each PSE service can be mapped to a specific tool that executes within an environment. For some of these services, the mapping is quite simple. The Compilation service obviously maps to the Ada or FORTRAN compiler in the NASA SEL environment. However, other services of the PSE model map to process steps in the NASA environment. For example, there is no single software testing tool in the FDD. Instead, the software testing service becomes a process step enacted as a sequence of tasks, some manual and some automated. We wish to characterize such processes.

In what follows, we use the classification of services in the PSE reference model as a means to characterize the functionality of tools that may appear within an environment. Our goal is to show the relationship between these services and the set of process steps that define the software development process.

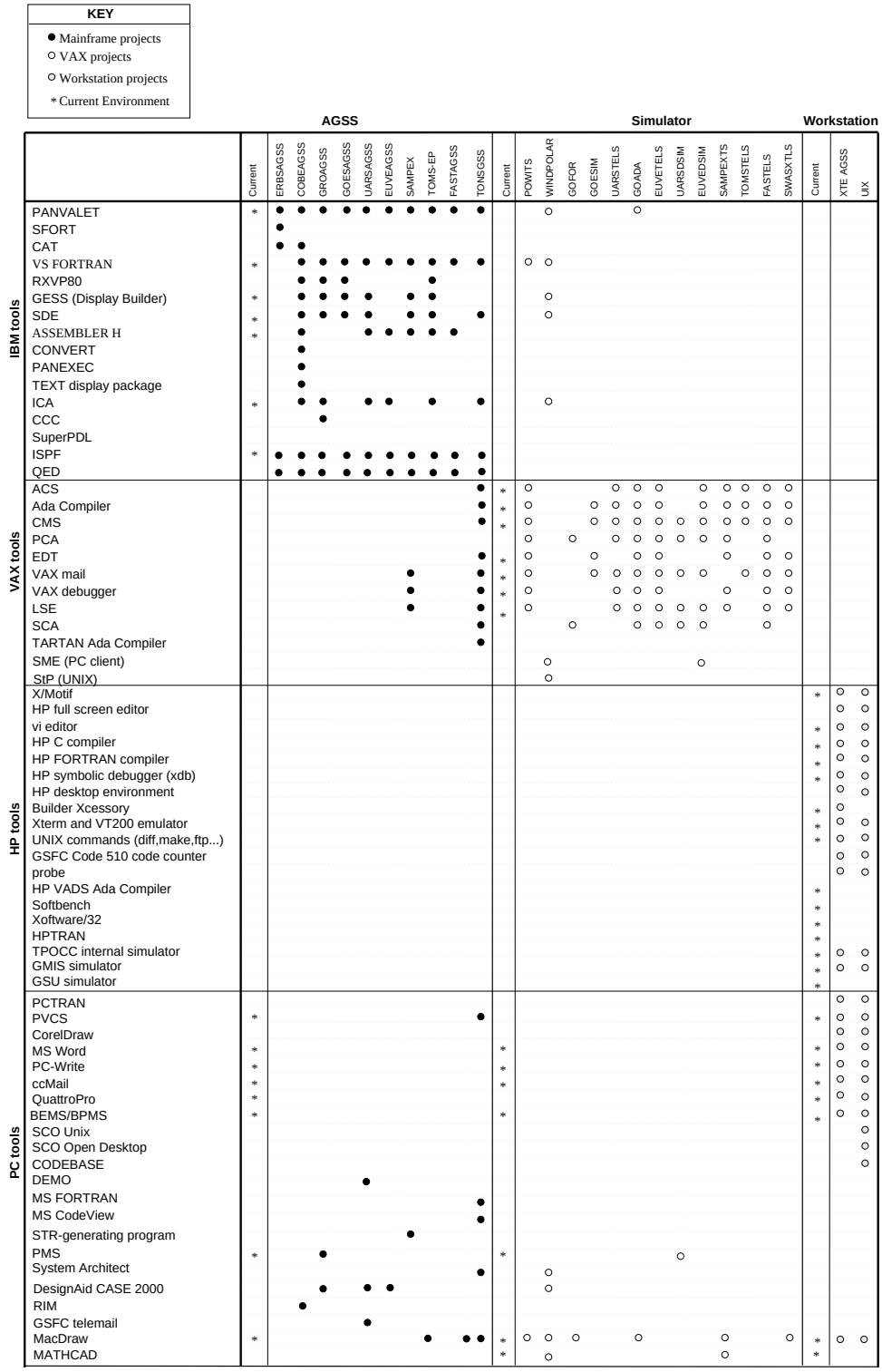


Figure 8: FDD tool use.

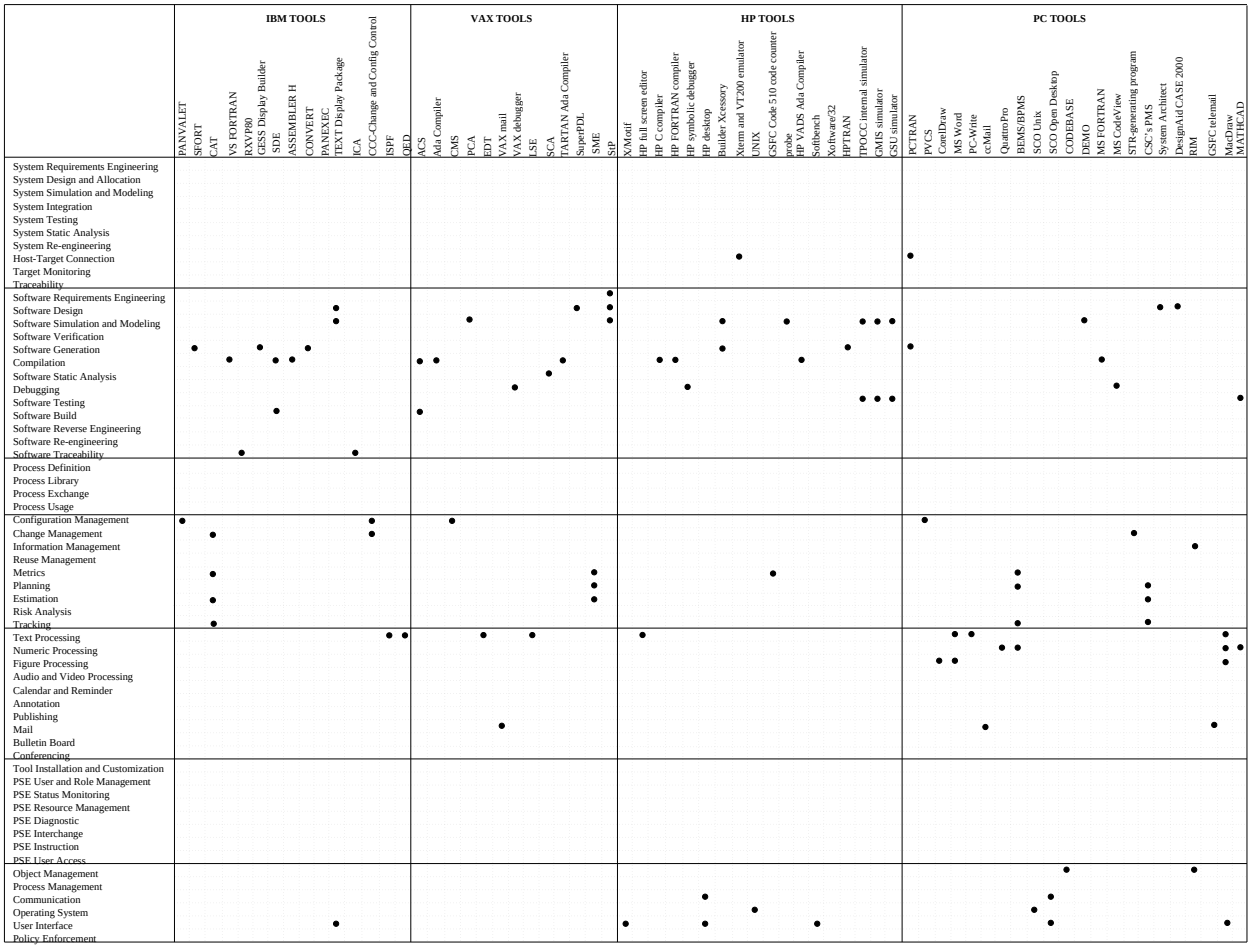


Figure 9: FDD automated tasks.

### 4.3.2 FDD Process Enactment Examples

To understand how tools are used in the FDD environment, it is instructive to examine how processes are enacted in the environment. Services can be provided in various ways. For example, a tool set could be integrated to present the user with the necessary services. Integrated toolsets such as Microsoft's Office (Word, Excel, Powerpoint and Access) and IDE's Software through Pictures are commercial examples of this. In such cases the user interacts with a single interface to ease the transition among the various tools in the colleciton. As mentioned earlier, the SEL developed the SDE interface for the mainframe environment in order to provide this integrated set of tools to the developer.

Alternatively, the user may have to manually change among different tools to obtain the required services. In what follows, we examine examples of the enactment of several processes in the FDD environment.

#### New Module Development

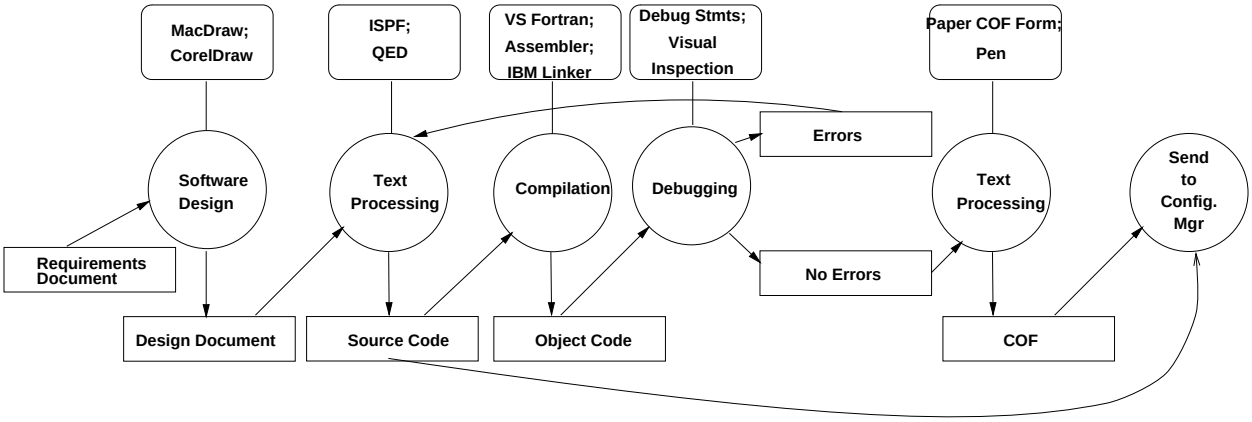


Figure 10: New Module Development Subprocess in the Mainframe Environment.

The Level 1 Services have been defined to be the most fundamental services of software development, without which software cannot be developed. Consider the process steps for creating new software modules in the mainframe environment. Most of the Level 1 Services are represented in this process. Starting with the module's requirements, the developer must design the module, implement it in FORTRAN, test it, and then release it for configuration control. This involves a complex series of interactions using several tools on the mainframe. This is given by Figure 10. The key for Figure 10 is presented in Figure 11.

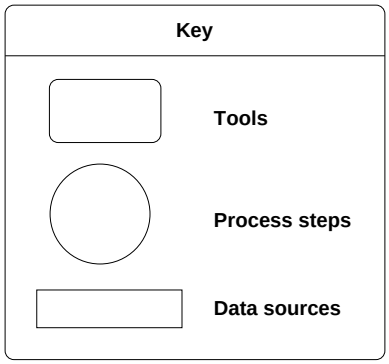


Figure 11: Key for process diagrams.

Several interesting aspects are demonstrated by this figure:

1. No design tool is employed in the Software Design process step. A standard picture-drawing program is used for design documents. This limits the ability to automate traceability back from source programs to specific requirements.
2. The Text Processing process step is employed twice in this process. In one case, an editor (e.g., ISPF) is used to create and modify the modules, while in the other case, a pen is used to complete a paper form for processing the Component Origination Form (COF).
3. There is no debugger available in this environment. To compensate for this, debug statements must be placed in the code and the programmer must run the program to find errors. Notice that debugging is defined by us as a Level 1 Service; however, it is not achieved easily in the mainframe environment.

4. Each activity of the new module development process requires the user to explicitly request the services from each tool. There is no automated sequencing from one activity to the next. For example, there is no guarantee that the design is complete before the Text Processing activity begins or that no errors are found in testing before module is released for Configuration Management. This is not to imply that the developers "cut corners," only that there is no automated process for ensuring that quality control aspects of the process are followed.

As we stated earlier, the FDD is now achieving much higher reuse rates in their software development. How is that achieved? Later we give the Software Reuse process, and indicate how it differs from new module development.

**Software Testing**

The FDD projects have good reliability rates, yet there are no standard tools available for testing. Typically, previously developed simulators are used to generate and send data to test the new systems.

One explanation for the high reliability in FDD systems may be familiarity. Since the systems constructed by the FDD are very similar in functionality, new systems benefit from the testing done on previous systems. In some cases, up to 90 percent [1] of the functionality is supplied by an existing system. The FDD has become very good at developing the types of systems where they have experience.

When a system on a completely different architecture was attempted, the results were different. There were many problems with the workstation projects [4]. The developers were not prepared for the changes required for a completely different architecture than the ones of previous systems.

**Reuse Management**

When looking at reuse, two different approaches can be identified:

- Reuse achieved by adjusting the components of an existing system to fit the requirements of a new system.
- Reuse achieved by taking components from various sources and piecing them together to form a new system.

The process for module reuse in the mainframe environment is presented in Figure 12. The most notable feature of this figure is the Reuse Management step of the process. In the FDD, reuse is highly dependent upon the personal knowledge of the designers. Component reuse is achieved by the designer verbally communicating the name of the component to the programmer. Although this does not appear to be an effective approach to reuse management, the FDD has extremely high reuse rates. There are several possible explanations for these results. The FDD tends to use the first of the two approaches mentioned for reuse. The functionality provided by new systems in the FDD are very similar to the functionality of existing systems. Therefore, the code from the components of an existing system can be used verbatim or with slight modifications in a new system. This type of system construction does not require a sophisticated reuse system or process. A designer is not likely to forget about a component from one project to the next. This approach does, however, require a configuration management tool. That could explain why, the existing project support environments of the FDD do not include any reuse tools or reuse mechanisms, but

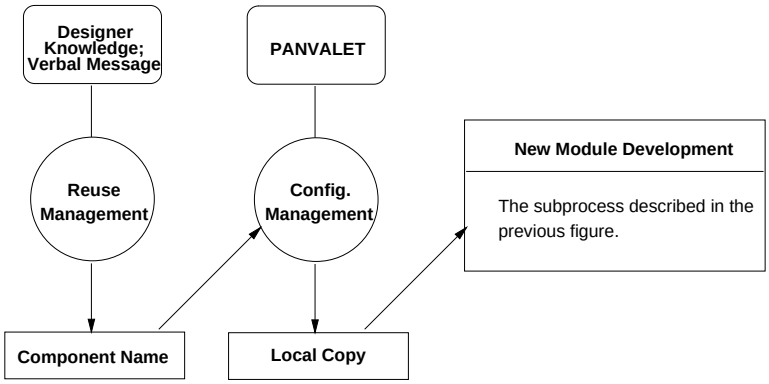


Figure 12: Module Development from a Reusable Component in the Mainframe Environment.

do have several configuration management tools. It also explains the extremely high reuse rates of 80 to 90 percent [1] for the projects.

On the POWITS project, the reuse rate was not as high as previous projects; it was 40 percent [1]. The dip in the reuse rate is attributed to a change in the domain. The simulation software had to be changed from a three-axis stabilized spacecraft to a spin-stabilized spacecraft. To accommodate the change, new software had to be written.

# 5 How do we classify flight dynamics application functions?

The final step in this development is to apply a measurement framework to our ITEM model. This workis only preliminary and addresses how we intend to continue this activity.

Our ITEM model depents upon three parameters: abstraction level, automation level, and process complexity. The following is a brief overview of how to impose a metric on top of these concepts.

As shown in Figure 10, a process is simply a graph, where the circles represent the services of the PSE model. As given in Section 3.2, we can classify the complexity of each service. We will define the absstraction level as the set of services from the PSE model that are addressed by the activity. A previously-developed graph complexity model [13] based upon an information theoretic view of program graphs, we can impose a measurement on each such activity. This represents process complexity. For automation level we have the degree of reuse of a given project since this represents the amount of effort that is not being undertaken on a given project. We can then correlate the point in the ITEM model defined by these three numbers with the number of errors or effort expended in developing a given product.

To complete this validation, we need to develop activity graphs for all SEL development processes and then for each project in our database, we need to plot its location on the model. While this will not provide an absolute scale for this model, it should provide a relative evaluation of different developments.

# 6 Conclusions

After studying this environment, we can separate our conclusions into those that reflect on software development within the FDD at GSFC and on conclusions reflecting process and environment research in general.

## 6.1 FDD Observations

1. *The overriding concern in this environment is the process for software development and not in the tool support for the developers and managers.* There is considerable care in developing models of how personnel interact. Process-centric methods for development, such as cleanroom and object-oriented technology, have been studied in depth.

2. *When the need for the automation of an activity arises, the FDD tends to develop a tool or search internally for a solution.* The use of SDE in the mainframe environment to achieve a level of tool integration is an example of this. Another example is the use of BPMS/BEMS spreadsheets for project status tracking and estimation. This approach led to difficulties in the transition from the VAX and IBM mainframe environments to an open systems workstation environment.

3. *The environments of the FDD have remained relatively constant over the years.* The FDD does not experiment very often with new tools in their environment. PANVALET has been used for configuration management in the mainframe environment for over 12 years. When a new tool is introduced, it is not given a "fair" chance. For example, StP was only tried one time. The users did not understand the methodology behind the tool. Instead of providing more training, the tool was not used again. This approach is very different from the way in which process-centric methods (e.g. Cleanroom) have been introduced over the years.

4. *Many of the process steps are enacted manually.* For example, there are numerous forms (Component Origination Form, Software Trouble Report) that are required by the SEL to track the software process. Currently, these forms are filled out with pen and paper.

5. *The testing process in the FDD does not utilize conventional software testing tools.* Instead, simulators designed and implemented by the FDD are utilized to generate data for testing. The rest of the testing process is mostly unsupported by tools.

6. *The software design process in the FDD doesn't use conventional design tools.* General purpose drawing tools are used to create design diagrams. This may lead to difficulties in software traceability.

## 6.2 General Observations

1. *We still need to impose our meaurement framework on top of the ITEM model in order to determine the effectiveness of our measurement approach.* This work is ongoing and will be reported on soon.

2. *The PSE reference model seems to be lacking a level of services.* Although the PSE reference model was useful for examining the coverage of the tools in the environment, it seemed to be missing a level of services. The reference model includes the concepts of infrastructure services as well as complex services (e.g. Software Design). There seems to be an intermediate level of services missing from the model. What are the process steps required for a complex process like software design? The current version of the reference model does not address this issue.

3. *The impact of reuse management tools is unclear.* FDD has high reuse rates, but utilizes no reuse management tools. What is the real impact of reuse management tools? We believe that we have identified two classes of reuse:

   (a) Create a system by reusing components from a similar system.

   (b) Create a system by using components from various systems or libraries of components.

   When discussing reuse, both concepts are usually merged, but they represent very different concepts. In the FDD case, as described earlier, reuse occurs by beginning with the reusable components of an existing system. Perhaps, reuse management tools are only necessary in development projects that assemble components from various sources.

4. *The impact of conventional testing tools is unclear.* FDD has very low error rates, yet uses no specific tools for testing. What is the real impact of testing tools?

## References


[1] Bailey, J., Waligora, S., Stark, M., Impact of Ada in the Flight Dynamics Division: excitement and frustration. In Procedings of the $18^{th}$ Annual Software Engineering Workshop, (Dec, 1993), 422-438.

[2] Basili, V.R., Caldiera, G. and Rombach, H.D., The experience factory. *Encyclopedia of Software Engineering*, Wiley, 1994.

[3] Basili, V.R., Caldiera, G., McGarry, F., Pajerski, R., Page, G. and Waligora S., The Software Engineering Laboratory – an operational software experience factory, ACM/IEEE $14^{th}$ International Conf. on Soft. Eng., Melbourne, Australia (May, 1992), 370-381.

[4] Boland, D.E., Green, D.S., Steger, W.L., Lessons learned in transitioning to an open systems environment, $19^{th}$ NASA Software Engineering Workshop, Greenbelt, MD (December 1994), 191-202.

[5] Brown A., Carney D., Oberndorf P. and Zelkowitz M. (Eds), The Project Support Reference Model, Version 2.0, National Institute of Standards and Technology, Special Publication SP 500-213, (November, 1993) (Also CMU/SEI TR 93-TR-23, November, 1993).

[6] Feiler, P.H., Humphrey, W.S., Software process development and enactment: concepts and definitions. In Proceedings of the $2^{nd}$ Int. Conf. on Software Process, Berlin, Germany (March, 1993), 28-40 (Also CMU/SEI-92-TR-4, 1992).

[7] Lonchamp, J., A structured conceptual and terminological framework for software process engineering. In Proceedings of the $2^{nd}$ International Conference on the Software Process, Berlin (February, 1993), 41-53.

[8] Kaiser, G.E., Rule-based modeling of the software development process. In Proceedings of the $4th$ International Software Process Workshop, Devon, UK (May, 1988), 84-86.

[9] Kaiser, G.E., A bi-level language for software process modeling. ACM/IEEE $15^{th}$ International Conf. on Soft. Eng., Baltimore, MD (May, 1993), 132-142.

[10] Kaiser, G.E., Feiler P.H. and Popovich S.S., Intelligent assistance for software development and maintenance. IEEE Software 5(3):40-49, May, 1988.

[11] NIST, Reference Model for Frameworks of Software Engineering Environments Special Publication 500-211, Natl. Inst. of Stnds and Tech., (August, 1993) (Also ECMA TR/55, Edition 3, (June, 1993)).

[12] Paulk, M.C., Curtis, B., Chrissis M.B. and Weber C.V., The Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, (1993).

[13] Tian J., and M. V. Zelkowitz, A formal model of program complexity and its application, *J. of Systems and Software* 17, 3 (1992) 253-266.

[14] Zelkowitz M. V., Use of an environment classification model, ACM/IEEE $15^{th}$ International Conf. on Soft. Eng., Baltimore, MD (May 1993), 348-357.

[15] Zelkowitz M. V., Software engineering technology transfer: Understanding the process, $18^{th}$ NASA Software Engineering Workshop, Greenbelt, MD (December 1993), 450-458.

[16] Zelkowitz M. V. and B. Cuthill, Application of an information technology model to software engineering environments, *Journal of Sys and Software,* 1996, *(To appear)*.