USING THE SOFTWARE PROCESS MODEL TO ANALYZE A SOFTWARE PROJECT

by

I. SUE RANFT

B.S., Ohio State University, 1983

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

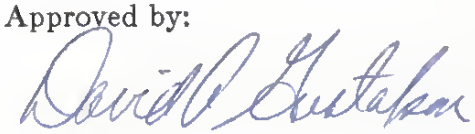
MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:



Major Professor

LD
2668
.R4
CMSC
1984
R36
c.2

A11208 317818

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. David A. Gustafson (Gus) for serving as the Major Professor for this Masters Report (and his wife, Karen, for serving the wine). Without his inspiration, ideas and reviews this report would not have been possible. His ability to motivate and keep me "pumped up" was not only appreciated, but greatly needed, and again, I thank him.

Thanks also goes to Dr. Elizabeth Unger and Dr. Masaaki Mizuno for serving on my committee. The hours they spent reviewing the report and providing constructive criticism were appreciated.

This report would not have been possible without the existence of the AT&T Summer-on-Campus Program. Therefore, I would like to thank AT&T for having the program and the Department of Computing and Information Sciences at Kansas State University for participating in the program.

During this program several AT&T students provided support and

encouragement. A few of these individuals deserve to be recognized. Due to Sue Saad's and Katryn Inkley's efforts, I was able to make it to the majority of my classes and appointments on time. Their continuous banging on my door, when conventional methods of waking up failed, was greatly appreciated. Without, the use of Sue Saad's dictionary and thesaurus this paper may have never been presented to my committee. Thanks, Sue, for letting me use them during the third shift. Thanks goes to James Watson, not only for keeping me out of trouble during my stay at KSU, but also for insisting that I stay in my room and continue writing when I wanted to participate in other activities. I must also thank Dave Huchro for the late night mini-breaks and mail messages of encouragement. When the mini-breaks were not enough, thanks to Debbie Schoonover, Scott Young and cab 30, longer breaks were provided.

Thanks also goes to the entire staff and faculty at Kansas State University, Department of Computing and Information Sciences. The educational experience will always be cherished and remembered. A special thanks to Sandy Randel for her help in formatting this report

and the making of vu-graphs (or as they are called at KSU, overhead transparencies).

Last, but not least, I want to thank my husband, Mark, and sons, Tucker and Elliott, for their encouragement and support. I also must thank them, not only for tolerating my absence from home for the past five summers, but also for tolerating my return.

1. INTRODUCTION

Software development has grown rapidly over the years. It has doubled in size approximately every five years and it is estimated that it will experience a tenfold increase each decade [MUS85]. Throughout its growth, it has experienced late deliveries, cost overruns and customer dissatisfaction. Although management and technology have been recognized as parts of these problems, the majority of research done has pertained to the technical side of the problem [ABD87].

Models of the software development process have been developed to provide a means of controlling and visualizing software projects. The most common software model is the Software Life Cycle or Waterfall Model [BOE81]. This model is depicted simply in Figure 1. The Software Life Cycle Model divides the development process into several sequential phases. Although, the number and names of each phase may differ from project to project, basic phases would include the requirements phase, analysis phase, design phase, coding phase, testing phase and maintenance. Associated with each phase is a set of distinct documents that will be produced and used as inputs for the next phase. The model also includes backward pointing arrows to indicate

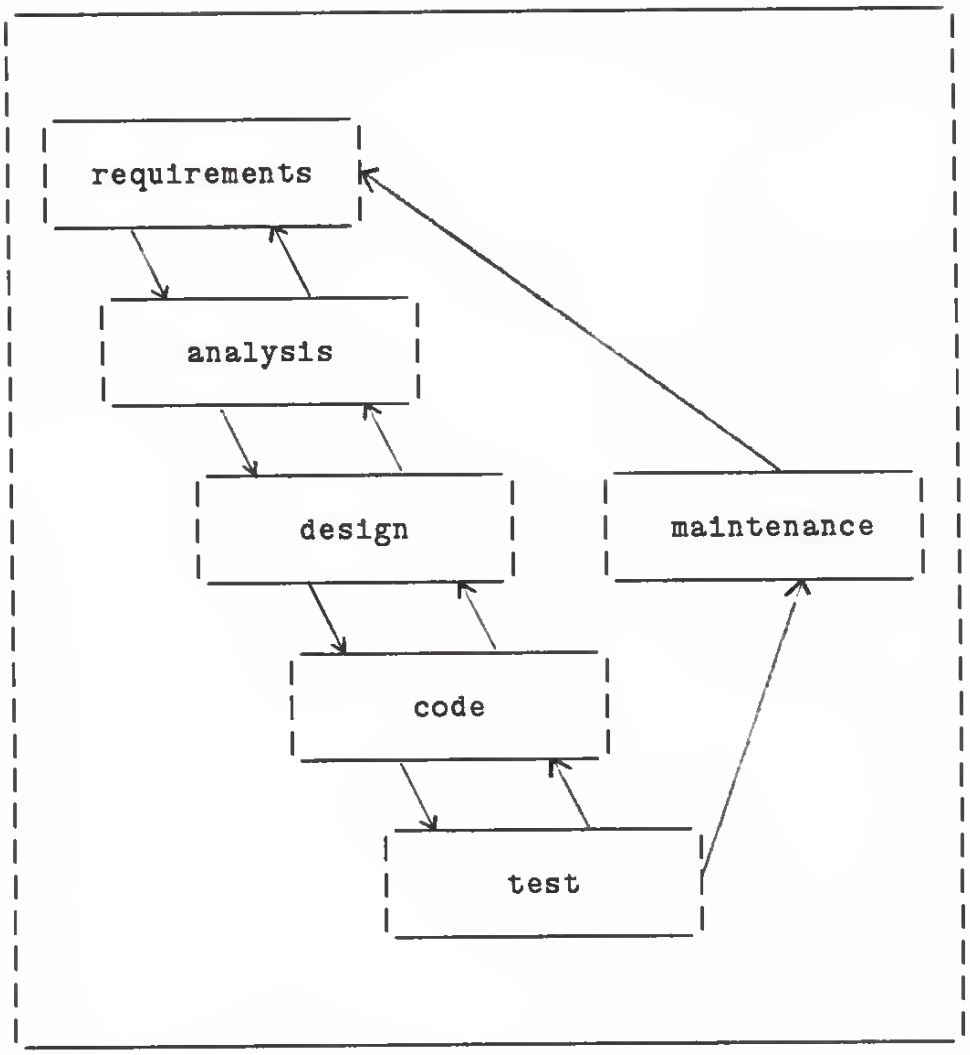


Figure 1. A Simple View of the Waterfall Model

backtracking to previous phases due to errors detected in later phases. This model has provided traditional project management tools such as

Gantt charts, Critical Path Analysis (CPA) and Project Evaluation and Program Technique (PERT) with a natural way of defining milestones and tasks. Although these management tools focus on scheduling activities and resource computation, they do not have the capahility to handle the iterative process of software development [LIU88]. Therefore, controlling and visualizing the true development process is difficult.

In 1987 another software model was introduced, the Software Process Model (SPM) [GUS87]. There are two major differences hetween the SPM and the Software Life Cycle Model. First, the SPM does not view software development as a simple, sequential process. Instead, the model assumes that development occurs in parallel. Second, the SPM models software development hy modeling the evolution of the full set of documents produced in a software process. These differences make the SPM general enough that it could he usahle for any project using any development approach.

This paper is the results of a study to evaluate the usefulness of the SPM in controlling and managing the software development process. This paper will describe the SPM and the software project to which it

was applied. It will then present information generated by reviewing the project from the viewpoint of the SPM and demonstrate the SPM's usefulness in managing a software project.

2. THE SOFTWARE PROCESS MODEL

2.1 OVERVIEW OF THE SOFTWARE PROCESS MODEL

The Software Process Model was introduced in 1987 in a paper titled "Modeling and Measuring the Software Development Process" [GUS87]. The information presented in this chapter has been taken exclusively from this article and another titled "The Software Process Model" [GUS88].

The SPM is a product-based model of the software development process. The product being defined is more than the executable implementation; it also includes documentation produced (both, formal and informal) during the development process. The SPM actually models the evolution of the documents produced as part of a software project.

Traditional software development models have the following deficiencies:

1. They prescribe a sequential order of document development.
2. They focus on final documents.

3. They do not include all the documents (documents expressed informally).

The SPM does not suffer from these deficiencies. In fact, the SPM assumes parallel development of documents. It also assumes that information learned from earlier and/or later efforts on a document is used as one works toward the final version of a document. Since the SPM does not prescribe an order of activities and there is no assumption made on the number, form and purpose of documents produced, the SPM may be applied to a variety of lifecycle schemes.

The SPM views the development of a software system as the process of transforming representations of the system. Each traditional phase of the software development lifecycle produces a distinct representation of the system. The SPM views these distinct views as documents and each provides a unique view of the system. Therefore, each document that represents these views must also be identifiably distinct. Using the SPM, each document is either written in a different language or the documents are identifiable in some other manner. Thus, a requirements document can be distinguished from a specification document; a specification can be distinguished from a design; a design

can be distinguished from an implementation; and so on.

2.2 FORMAL DEFINITION OF THE SOFTWARE PROCESS MODEL

Since the SPM views the software development as the process of evolving a set of documents, the process is time dependent and therefore a real-time clock is a critical component of the model. The following definitions formally characterizes the SPM.

1. DEFINITION:

   The Software Process Model (SPM) is a set of document histories, $SPM = \{H_1, H_2, \ldots, H_n\}$ where each H_i is a history of the different versions of one document.

2. DEFINITION:

   A document history Hi is a tree whose nodes are versions of documents, and $H_i = (V_i, E_i, r_i)$ where

   - V_i is a set of document versions of type i,
   - E_i is a set of order pairs of the form (a, b),

     $a, b \in V_i$, which represent transitions from one version in V_i to another, and

- r_i is the root of the H_i tree or the initial version.

The SPM uses a tree to represent document history to allow the modeling of the development of alternative final versions (e.g. development of a system for various customers with one or more different features).

3. DEFINITION:

A document version $d_i(j) \epsilon V_i$ is an ordered pair $d_i(j) = [d'_i(j), td_i(j)]$ where

- $d'_i(j)$ consists of the text of the document version, and
- $td_i(j)$ is a real time stamp which represents the completion date/time for $d'_i(j)$.

The time stamp can he augmented with additional management data which will allow the calculation of various measures such as the numher of person-hours used to develop the document versions.

The granularity of a document history is arbitrary and depends upon the needs of a particular project. One can either determine

that a new version is created when a developer reports that a document is ready for review, after the review and required corrections made, or each time the online version is edited.

The use of the SPM to describe a possible software project is illustrated in Figure 2. This represents an idealized development that went through two passes on the documentation.

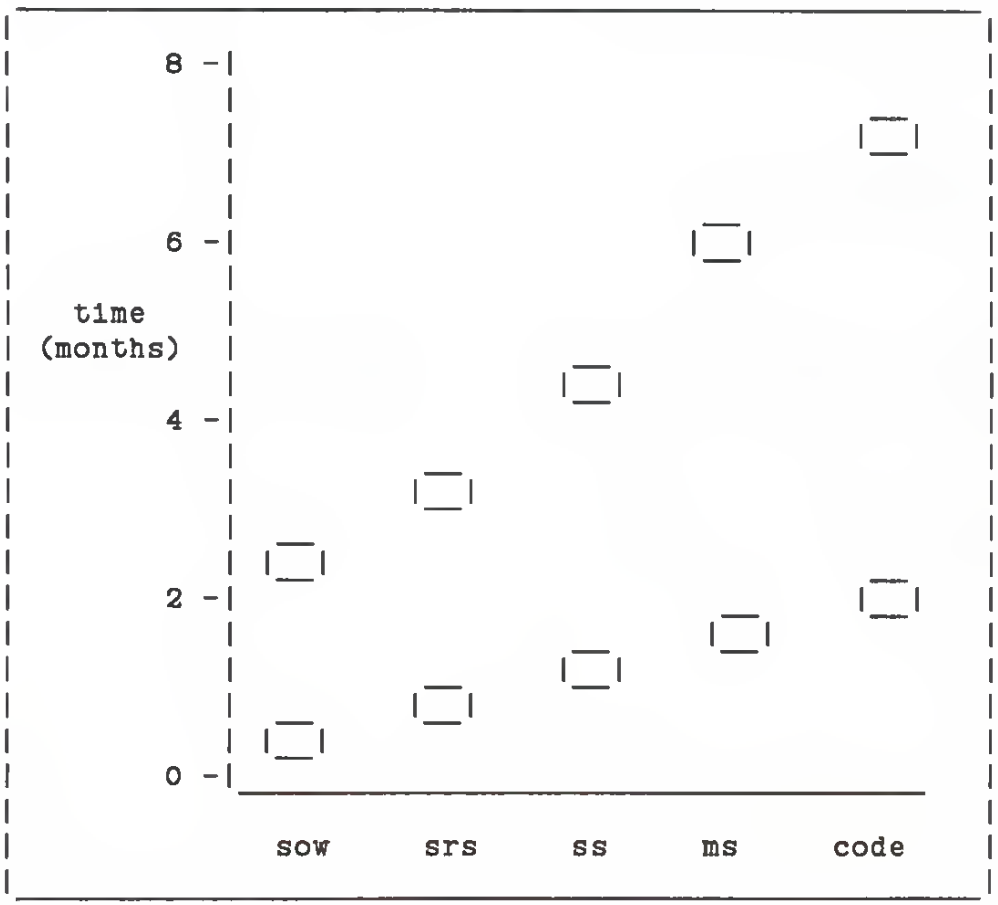


Figure 2. SPM for an idealized development [GUS88]

The SPM shows patterns of software development in much greater detailed than the Software Life Cycle Model. In addition the SPM models parallel activities and provides a complete project history. The next chapter will describe a software development project in which the SPM was applied. It will be followed by a chapter describing how the SPM was applied.

3. THE SOFTWARE PROJECT

3.1 ORGANIZATION OF THE PROJECT

The goal of the software project was to develop an information system to replace an existing system which no longer adequately met the users needs and was costly to maintain. The existing system ran on an IBM mainframe, used a hierarchical data base which was no longer supported, and was written in PL/I. The system had little, if any documentation, which made maintenance virtually impossible. The users wanted the new system to be fully documented, run on their own AT&T 3B4000 machine, use INFORMIX database system and be coded in 4GL, C or embedded C.

The project's development team was staffed with personal from two locations, in different states, and the users were located in nine different states. This made management of, and communication within, the project difficult. The individuals and groups that were involved in the project is depicted in Figure 3 and a list of their responsibilities are listed below.

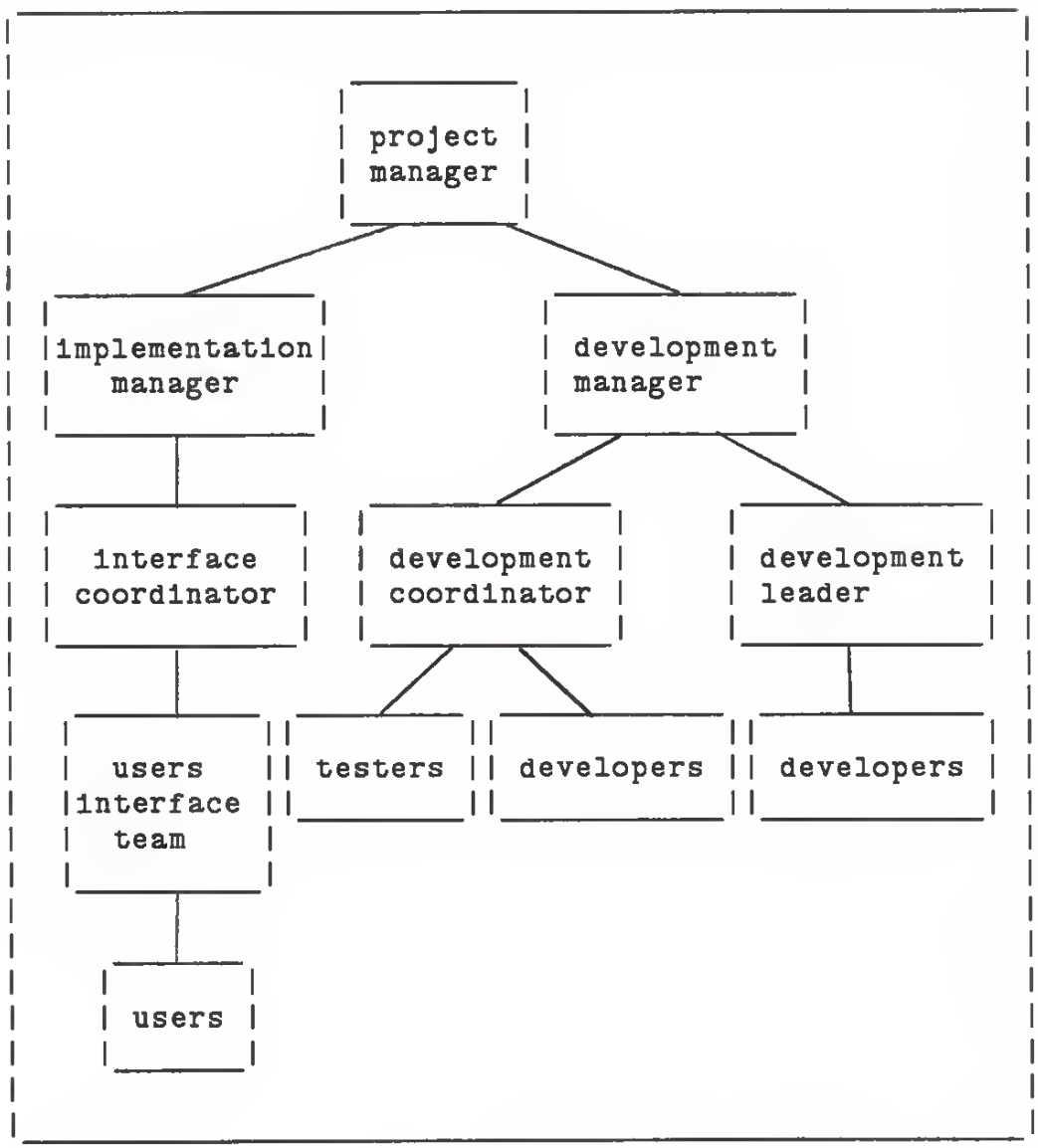


Figure 3. Organization Chart of Project

1. The project manager had to ensure the successful execution of all phases of the project to ensure its overall success. She was responsible for providing the necessary project planning, had to coordinate and integrate activities across multiple functional lines, control all project funding and staffing requirements, assign and track action items, monitor progress, ensure that adequate interfaces were in place to disseminate information and provide it in a non-disruptive manner, and coordinate project management status meetings.

2. The implementation manager was responsible for ensuring that the user interface team's activities were performed according to the agreed upon schedule. He acted as chairperson of the user interface team and had to keep the project manager informed of all user activities and status.

3. The development manager was responsible for ensuring that the software development activities were performed according to the agreed upon schedule, raising alerts when appropriate to ensure successful development of the software and keeping the project manager informed of all development activities and status.

4. The development coordinator was responsible for monitoring the progress of the various software development activities (in both locations), reporting the development status to the development manager and coordinating users' and developers' review meetings.

5. The development leader was responsible for coordinating and monitoring the software development efforts of the remote location, reporting the remote location's development status to the development coordinator and coordinating all remote review meetings.

6. The interface coordinator was responsible for coordinating the various user interface team activities and monitoring the progress of these activities, reporting the status of implementation activities to the implementation manager and acting as the primary contact for the developers.

7. The developers consist of personnel from two locations. The responsibilities of this group was to provide the software development and to review the user documentation.

8. The testers are responsible for developing system test plans and

performing system testing.

9. The user interface team was composed of one user representative from each location and was chaired by the implementation manager. They were responsible for ensuring that the users' needs were communicated to and understood by the developers, and that the developers' needs were communicated to and understood by the users. They had to participate in analysis and design reviews and to ensure that specifications were understood by users at their locations. They had to develop user acceptance test plans and perform the user acceptance testing. They also had to develop user documentation that would provide detailed information to the users on how to use the system.

3.2 PROJECT'S DEVELOPMENT METHODOLOGY

The software development followed the Software Life Cycle Model. Development was divided into five major phases, Analysis Phase, Design Phase, Implementation Phase (coding), Testing Phase (system testing and user acceptance testing), and Conversion Phase (parallel testing). Figure 4 shows each phase with their associated inputs and outputs.

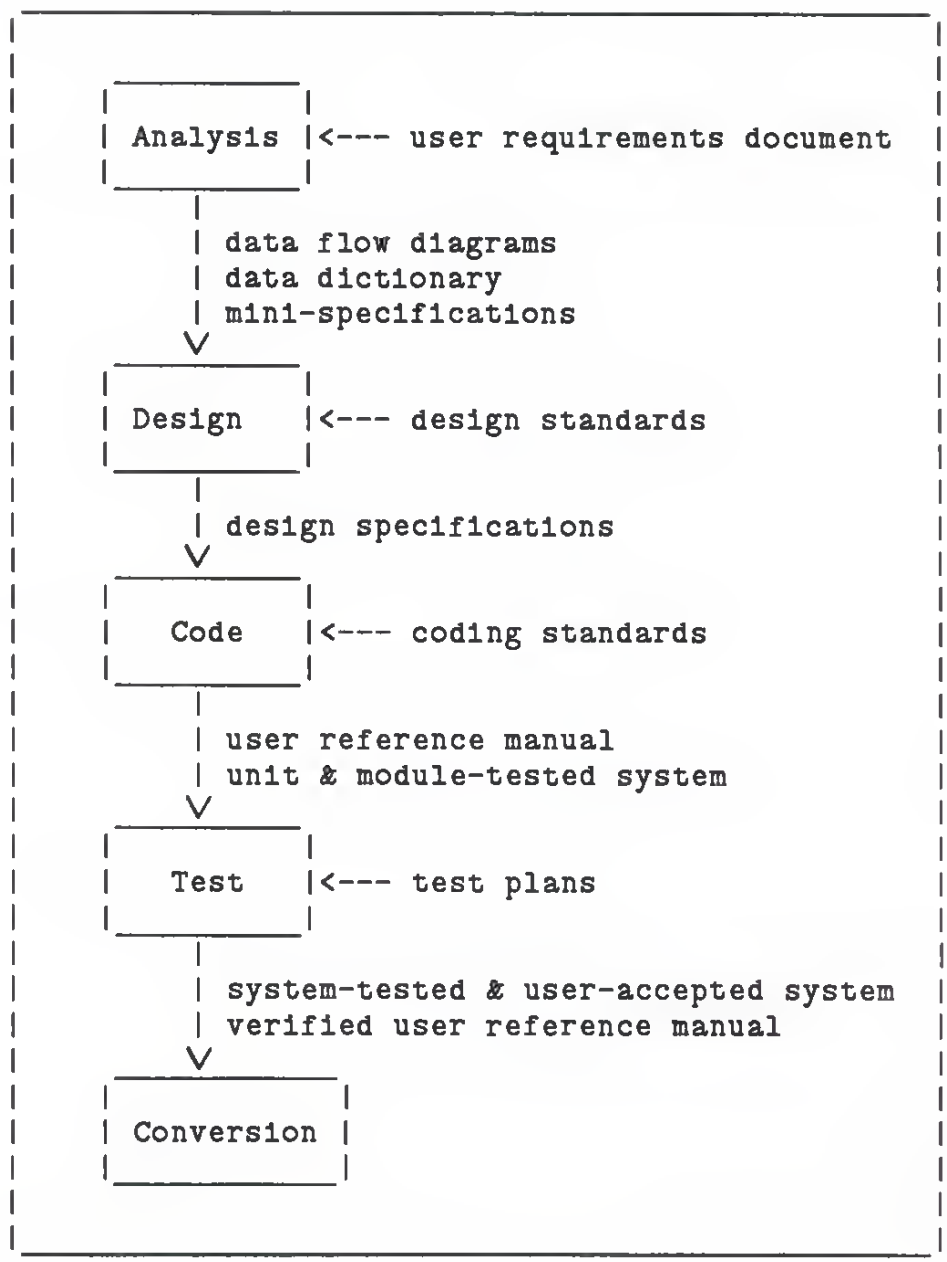


Figure 4. Phases of the Project

The Analysis Phase would begin after the user interface team had completed the User Requirements Document. During this phase the developers would analyze the required functions of the system. This would be accomplished by reviewing the user requirements with the user interface team. The input into this phase is the User Requirements Document and the outputs are data flow diagrams, data dictionary and mini-specifications.

The Design Phase would begin after the documents from the Analysis Phase had been approved by the user interface team. Activities in this phase involves reviewing the mini-specifications that were produced in the Analysis Phase and determining how to provide the system that was specified in these documents. Development of the test plans and user documentation would begin during this phase. The inputs for this phase are the data flow diagrams, data dictionary and mini-specifications from the previous phase and design standards. The outputs of this phase are the design specifications.

The Implementation Phase (or coding) would begin after the design specifications had been completed. Activities involved translating the system design specifications produced in the Design Phase into

structured code. Each program would be tested separately upon completion. After every program in an entire module was tested, the module would be tested. This incremental approach to program testing was to ensure that errors were detected early enough to make the necessary corrections and retest without affecting the entire system. The development of user documentation, the system test plan, and user acceptance test plan would also be completed during this phase. The inputs to this phase would be the design specifications and coding standards. The outputs would be the user reference manual and unit and module-tested system.

The Testing Phase would begin after the system software had been unit and module tested. This phase would include system testing and user acceptance testing. Activities included rigorously testing the completed system by using the system test plan that was developed by an independent group other than the developers. Other activities included verifying that instructions and information to perform the system's functions were included in the user documentation. The successful completion of this phase ensured the users that the system performs according to their expectations. Inputs for this phase

included system test plan, user acceptance test plan, user reference manual and unit and module-tested system. Outputs of this phase included system-tested system, user-accepted system and verified user reference manual.

The Conversion Phase (or parallel testing) would be the final phase. It would included operating the fully tested system and the current system in parallel mode until the users were comfortable with the performance of the new system. The new system would then cut over as the sole production system. Inputs into this phase are system-tested and user-accepted system and verified user reference manual. Output of the phase is a production system.

The use of the Software Life Cycle Model made creating milestones easy. Since it was anticipated that a phase would only start after the completion of the previous phase, management used the phases as milestones. Each milestone, or phase, was then divided into activities. The activities for each phase was virtually identical since they were associated with a particular function of the system. For example, the design phase would have an activity called "design function A", implementation would have a function called "code function A", and so

forth.

These milestones and activities, along with their expected start and completion dates, were then entered into a software program that generated bar charts. These charts were intended to provide management with a means of visualizing and controlling the project. However, they failed on both counts. Basically, the charts only showed if a particular activity or phase was on time or late. Even at that, the information would be erroneous, since the information was based on estimates of how much of the activity or phase was completed. Studies have shown that these estimates steadily increase from the beginning of a task until the activity reaches 80 to 90 percent completed. Then there is little increase in percentage of work done until the task is completed.

The next chapter will explain how the Software Process Model, when applied to this project, would have been a better management tool.

4. APPLYING THE SOFTWARE PROCESS MODEL

The SPM was applied to one of the twelve sub-system of this project. There are two reasons why this particular sub-system was chosen to apply the SPM. One reason was that this sub-system was one of the project's larger sub-systems. The other reason was, while most of the sub-systems had two or three developers working exclusively on them, this sub-system had many different developers working on different phases of development. If the SPM showed signs of peculiar patterns, then it was important that these patterns related to the development process not a particular developer's style.

To apply the SPM to this project a list of all documentation produced during the project was needed. Also needed, was the history of each document (e.g. initial release date and dates that changes were implemented). This information was then plotted on a graph with the x-axis representing time/date and the y-axis representing the documents. Added to the graph were any unresolved modification requests.

After the documents and unresolved modification requests were

plotted, apparent patterns were observed on the SPM. Hypotheses were drawn from these patterns and further investigation proved these hypotheses were correct. This chapter will discuss these patterns and hypotheses to illustrate that the SPM would be a valuable management tool. In the case of this project, the SPM gave more insight into this project than the current method used. To make the illustrations more effective, only excerpts of the SPM will be presented in this chapter. However, the entire project's sub-system's SPM can be found in appendix A. When viewing the SPMs use Figure 5 as a key.

Once the SPM was plotted, apparent patterns were observed. One of these obvious patterns is depicted in Figure 6. As Figure 6 illustrates, when the initial version of code was completed, the design specification was changed. Since this is only an excerpt of the original SPM, the statistics are not obvious. However, 18 of the 21 modules showed this pattern. That is approximately 86%. This percentage led to the hypothesis that the design specifications were not correct and had to be changed when coded. Interviews with some of the developers proved the hypothesis was correct. Their reason for inadequate design specifications was inexperience with structured design, which was the

required methodology in the design phase.

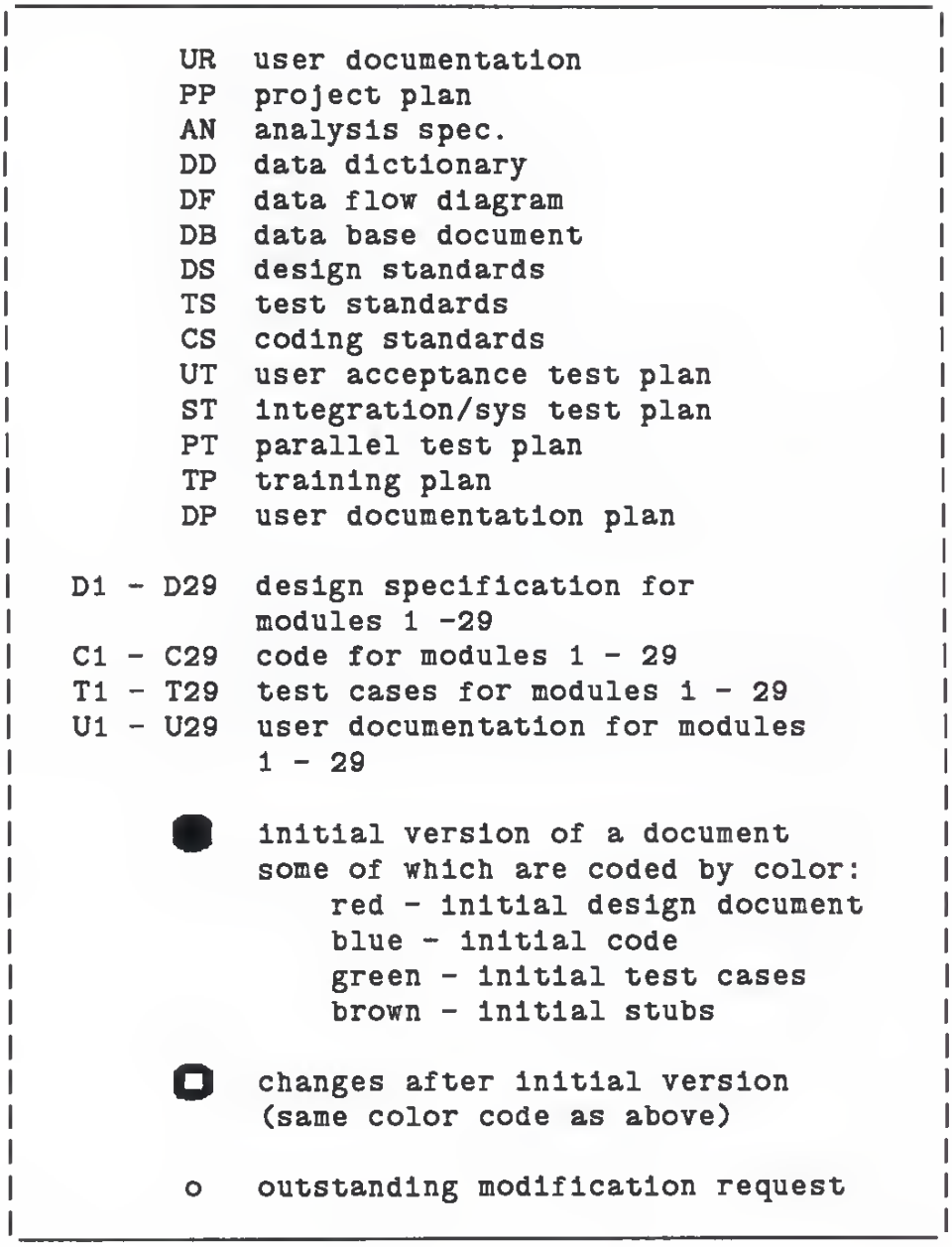


Figure 5. Key for SPM

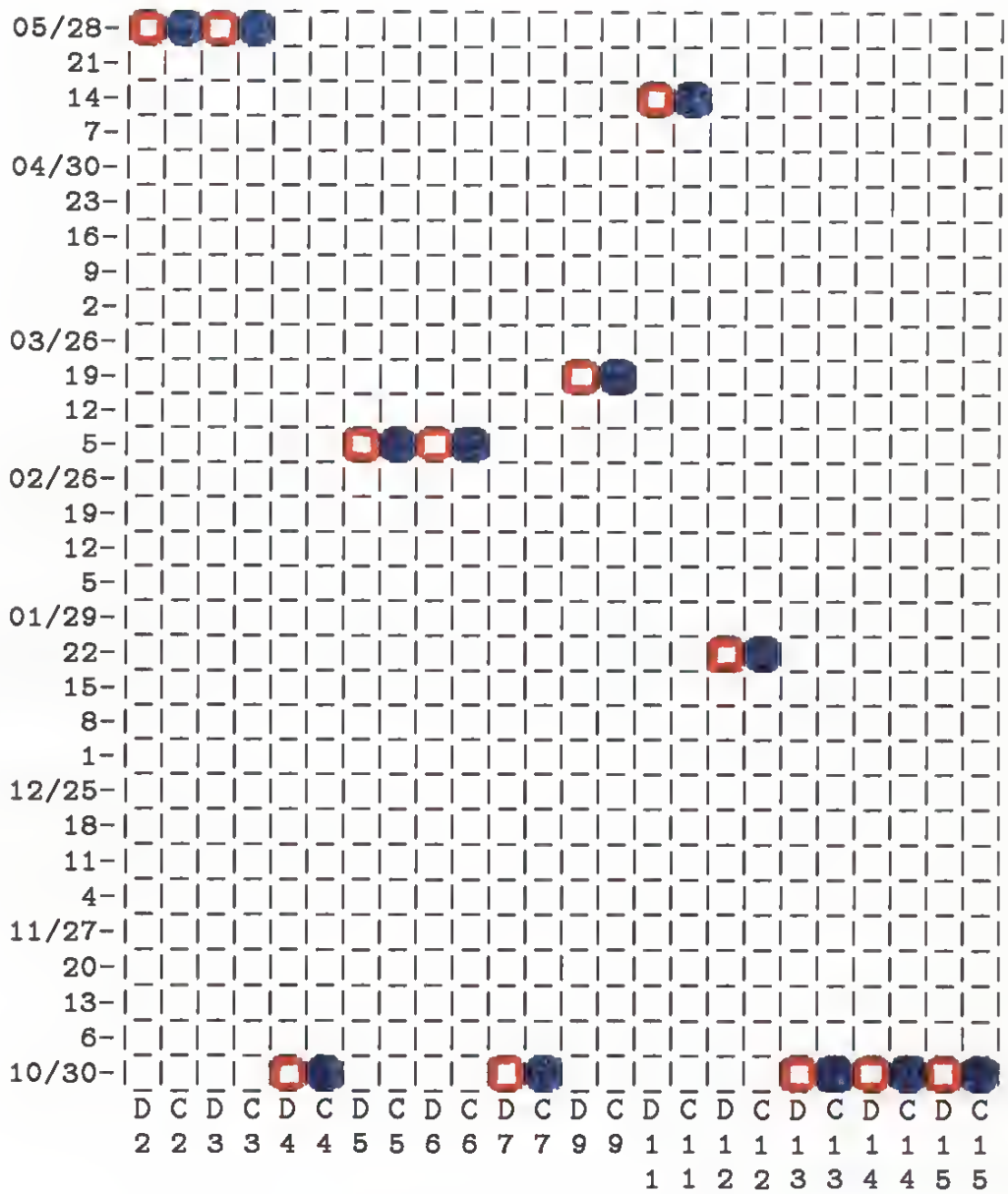


Figure 6. SPM - design and initial code relationship

Another obvious pattern occurred during the weeks of 1/8/89 and 1/15/89 and is depicted in Figure 7. In the first week approximately 24% of the design specifications and 24% of the coded modules were changed. In the following week the test cases associated with them were changed. Since these changes occurred during the same weeks it was assumed that they were caused by the same problem. I spoke with the developers and testers, and again the assumption was confirmed. The changes were due to the decision to handle error messages differently. The tester also further substantiated the findings of the first pattern. They indicated that 98% of the design specifications, for the entire project, were changed when the initial version of the software was released for testing. Due to these changes, the testers had to modify their test cases which were written from the design specifications.

Another hypothesis was made concerning the week of 1/29/89. During this week 11 out of 27 design specifications were changed. This is approximately 41%. Again a hypothesis was made that the changes were all related. Further investigation not only confirmed this hypothesis, but also revealed that these changes were due to the same

decision discussed above. This meant that 67% of the design specifications were changed due to the decision of handling error messages differently. The impact of this decision was astonishing.

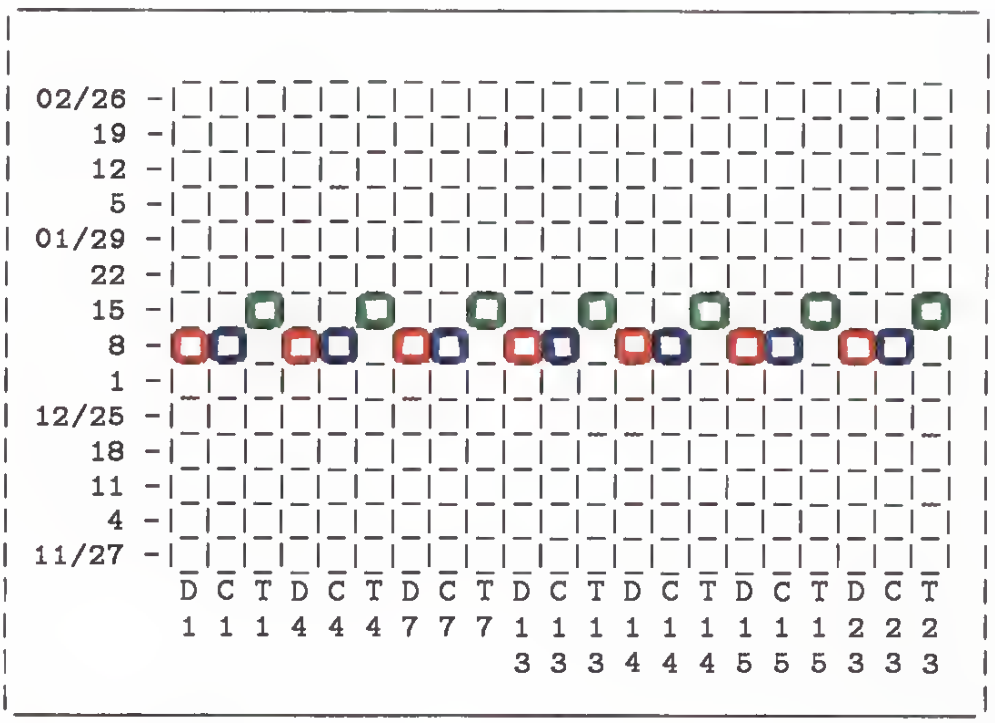


Figure 7. SPM - design, code and test relationship

During the week of 11/13/89, the SPM indicated that six of the ten coded modules (60%) were changed. Again, investigation proved that the changes were all related. Investigation also revealed that three of the four modules that were not changed, had been changed the week

before. This increased the percentage of modules changed to 90%. The remaining module which was not affected, turned out to be the cause of all the other changes. The SPM in Figure 8 illustrates these changes.

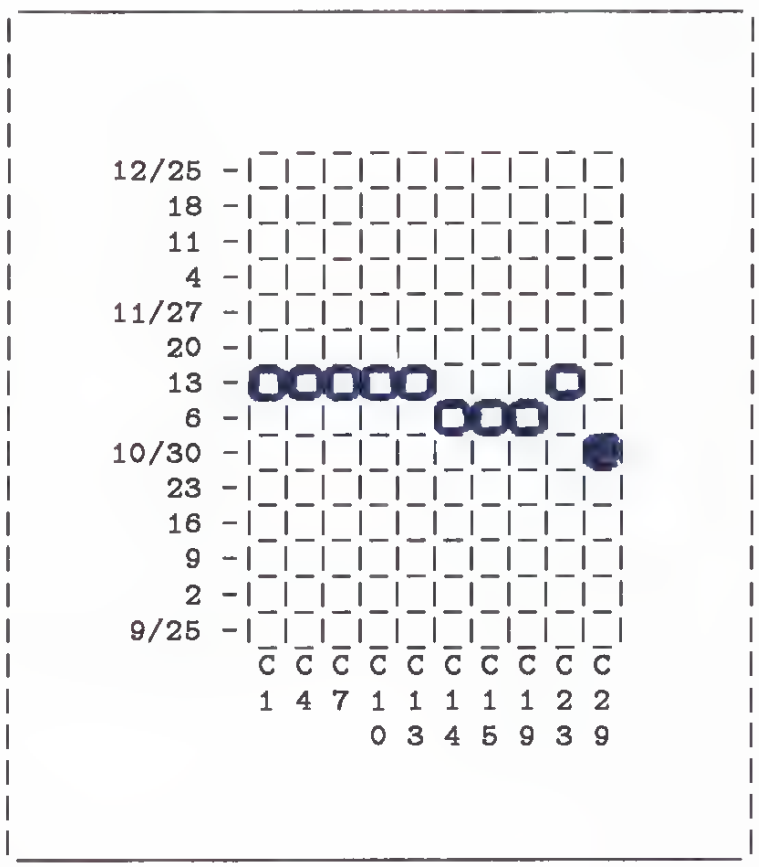


Figure 8. SPM - code changes

In reviewing the 1987 SPM it was evident that the analysis specification, data dictionary, and data flow diagrams were closely tied together. However, the relationship was not apparent in the SPMs for the next two years. Figure 9 contains an excerpt of the SPM for the three years. In 1987 the initial version of the three documents were released the same week. Also, in the same year, three changes were applied to the documents, again during the same time frame. Then in 1988 and 1989, two changes were made to the data dictionary, and none to the data flow diagrams or the analysis specifications. There were however, a number of changes requested for the analysis specifications and only one was requested for the data flow diagrams. From these SPMs, it was concluded that these documents were not kept up to date. It was further concluded that the modification requests probably affected more than the analysis specifications. Investigation into this matter revealed that these documents were not kept up to date. It was also discovered that 26% of the requested modifications for the analysis specification would also have to be reflected in the other two documents.

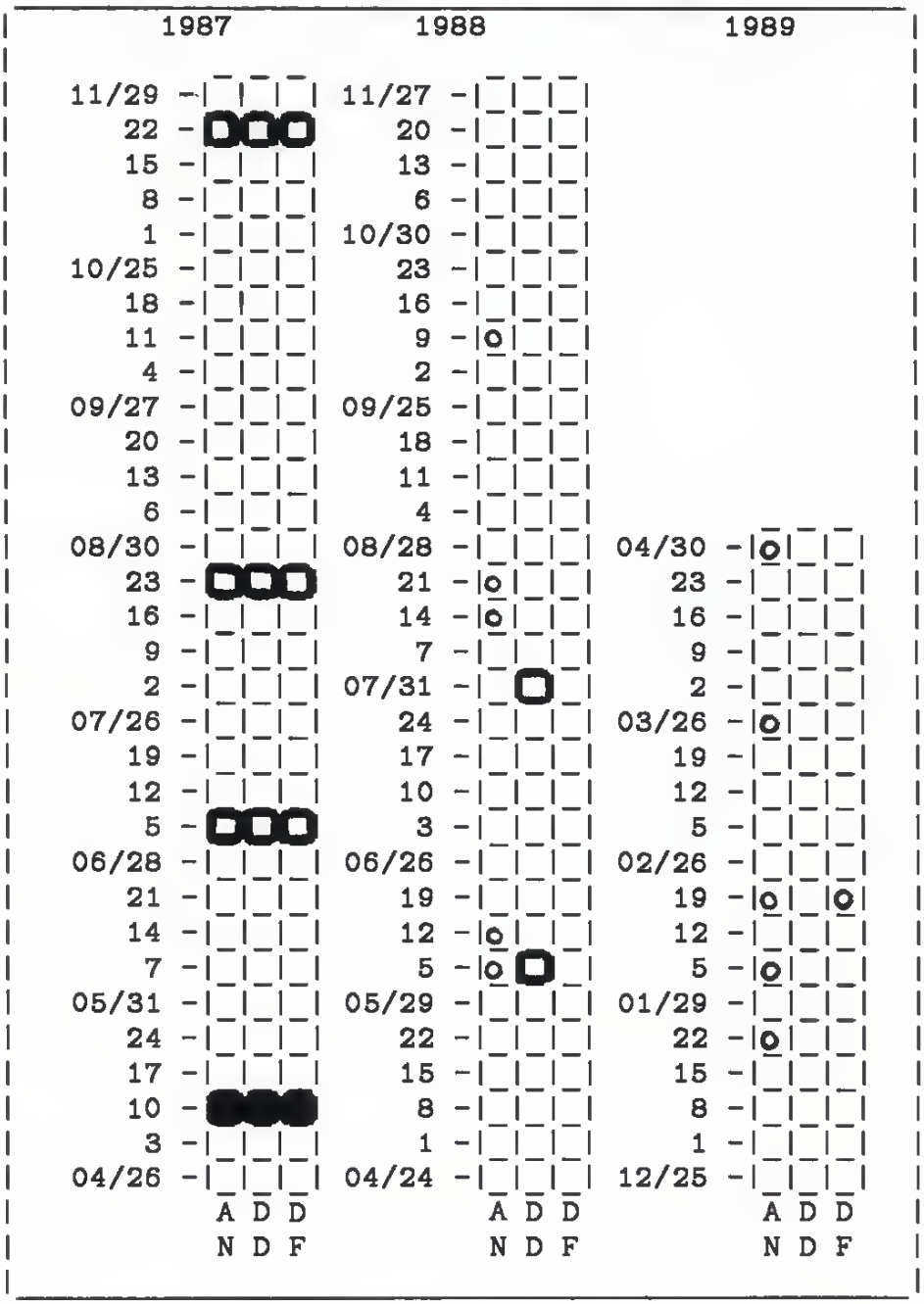


Figure 9. SPM - analysis spec, DD & DFD relationship

Figure 10 depicts yet another obvious pattern. From this illustration it was apparent that the design specifications were kept up to date to reflect changes made in the code. When this was investigated it was found that the developers were required to keep the two documents consistent and the testers tested for consistency.

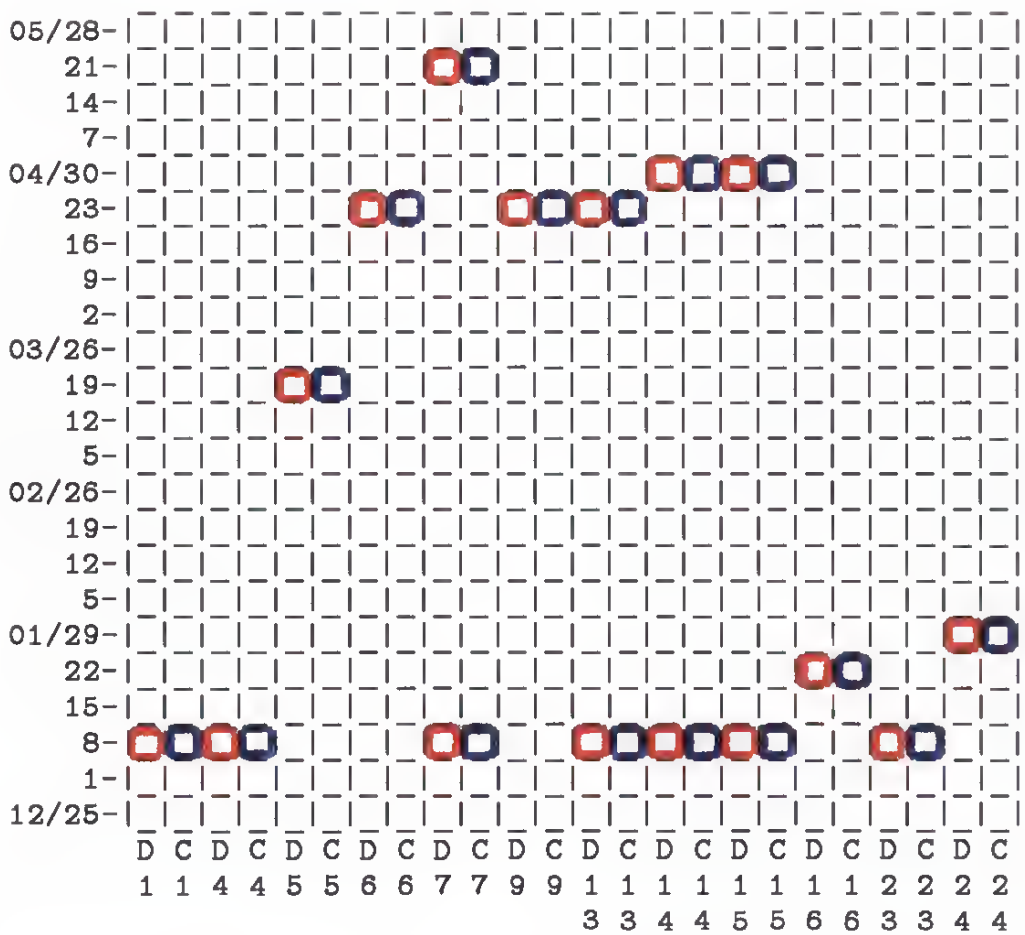


Figure 10. SPM - design specifications & code changes

5. CONCLUSIONS AND EXTENSIONS

This paper has demonstrated that the SPM can be a useful management tool. Since the SPM shows patterns of the software product development in greater detail then the current tools available, it provides management with a more effective means of visualizing and controlling software projects.

The SPM has the capability to alert management of problems encountered during the development process. This was illustrated when the SPM indicated a problem during the design phase. The problem was made known by the pattern generated by the continuous changes to design specifications when initial code was completed.

This paper has proved that the SPM can be used to ensure consistencies between documents. By plotting changes in documents, both developers and managers are alerted as to whether or not a change is incorporated in other documents. This will reduce, if not eliminate, inconsistencies between user's requirements, design and test specifications, user documentation and code.

The SPM shows parallel development effort and yet maintains the traditional correspondence between phases of development. Although

the software project presented in this paper used the Software Life Cycle Model and management set specific dates for the completion of each phase, the SPM indicated that the coding phase did not start after the design phase was completed. The initial code of some modules were completed during the week of 10/30/88 and other modules were not designed until the week of 12/22/89. Yet, at the same time, the SPM maintained the traditional phases for individual modules.

An extension to this paper would be automating the drawing of the SPM and developing an interface with a source control change system. The SPMs presented in this paper has proved that they have the ability to show the impact that a change has on the entire development process. However, no matter how great the impact, the information is after the fact. An interface to a source control change system could provide management with a tool to determine the impact of a change before the change is ever incorporated into a document. Whenever a change is requested a SPM could be plotted and indicate what documents that request would effect. For example, if the users requested that another functionality be added to a module then they would generate a modification request for their user requirements. If

that module was already coded and tested then the source control change system would generate two more modification requests, one for the code and another for the test cases. Before the modification request is accepted, an SPM could be plotted which would indicate the documents that had to be changed. This would enable management a means of visualizing the impact of the request before a decision is made to make to accept or reject the change request.

6. REFERENCES

[ABD87] Abdel-Hamid, T. K. and Madnick, S. E., "An Integrative System Dynamics Perspective of Software Project Management: Arguments for an Alternative Research Paradigm", MIT Industrial Liaison Program Report, April 1987.

[BOE81] Boehm, B. W., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GUS87] Gustafson, D. A., Melton, A. C., Baker, A. L. and Bieman, J. M., "Modeling and Measuring the Software Development Process", Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences, 1987.

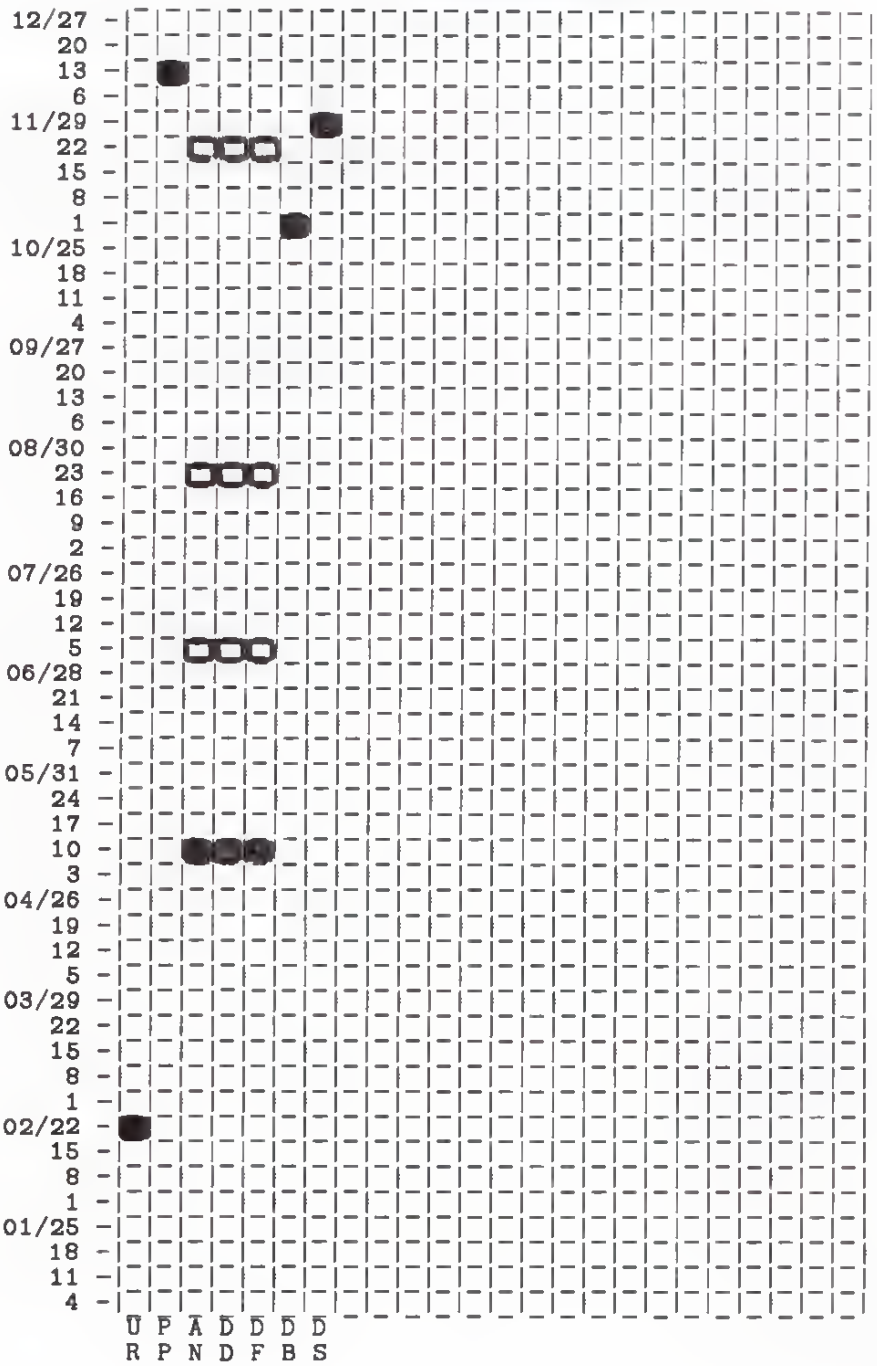
[GUS88] Gustafson, D. A., Melton, A. C., Chen, Y., Baker, A. L., and Bieman, J. M., "The Software Process Model", Preceedings from the Twelfth Annual International Computer Software and Applications Conference, published as CompSAC88, October 1988.

[LIU88] Liu, L. and Horowitz, E., "Object Database System for a Software Project Management Environment", Preceedings of the AMC Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, published as Sigsoft Software Engineering Notes, Vol. 13, No. 5, November 1988.

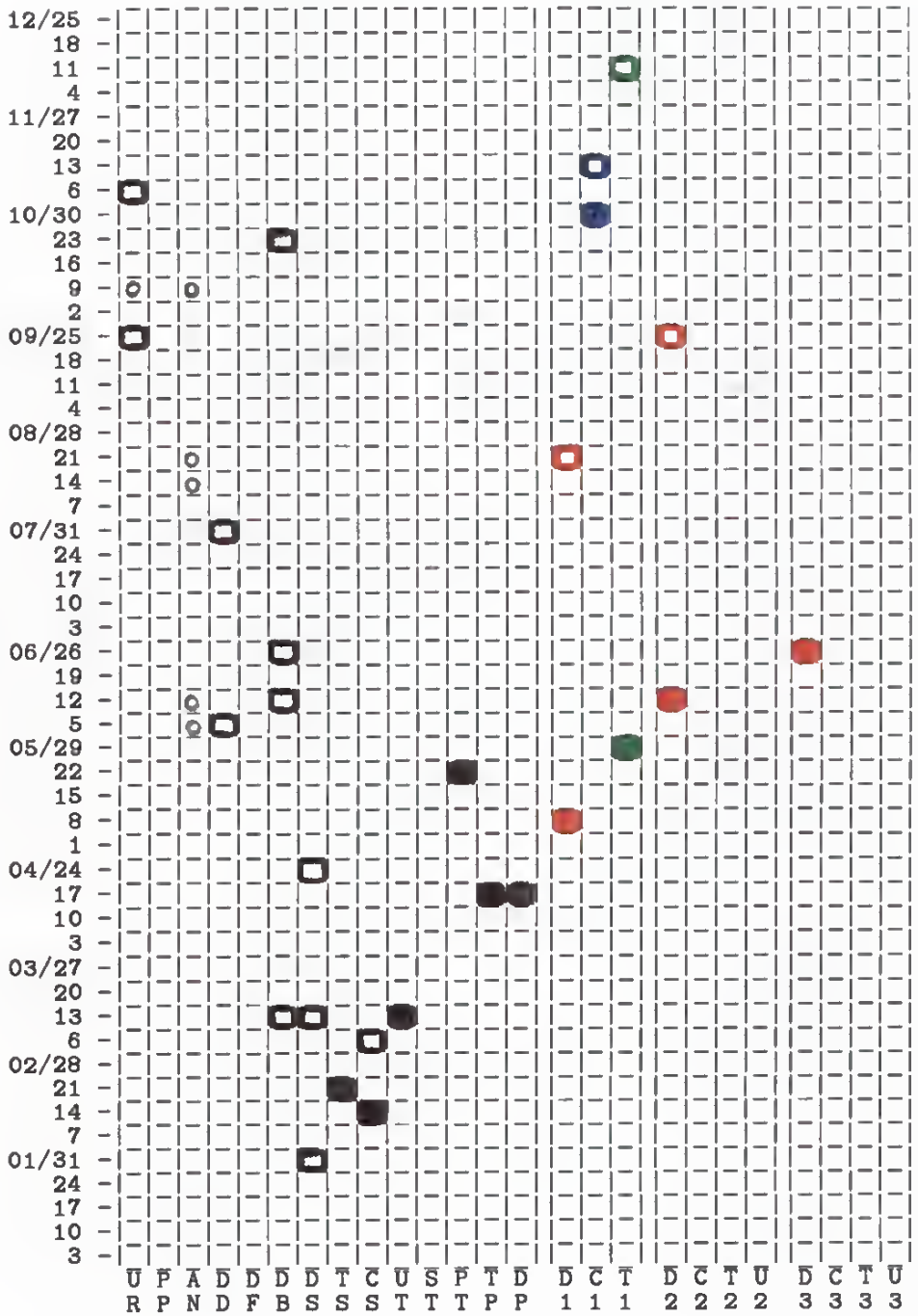
[MUS85] Musa, J. D., "Software Engineering: The Future of a Profession", IEEE Software, January 1985.

7. APPENDIX A

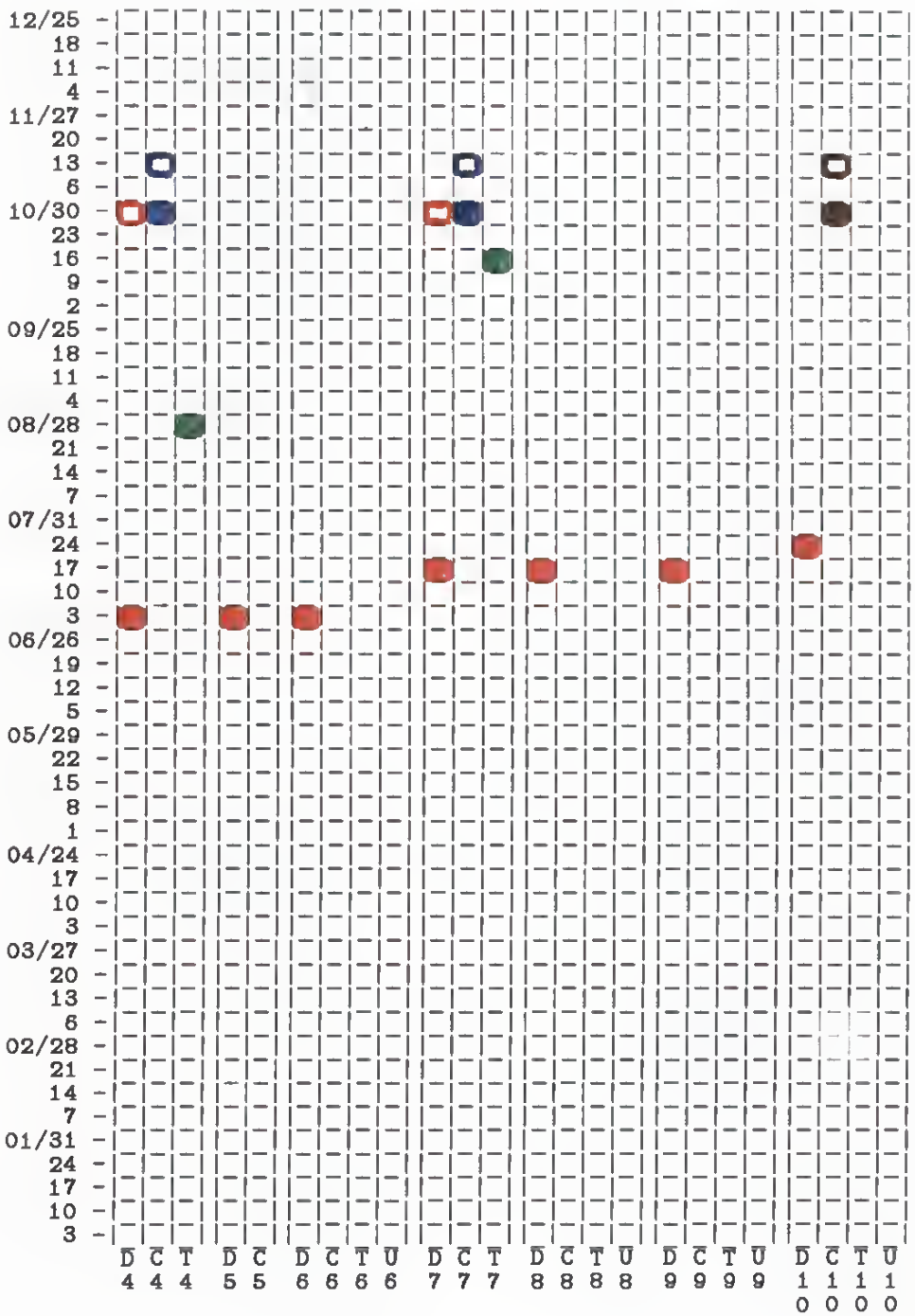
SPM FOR DOCUMENTS PRODUCED IN 1987



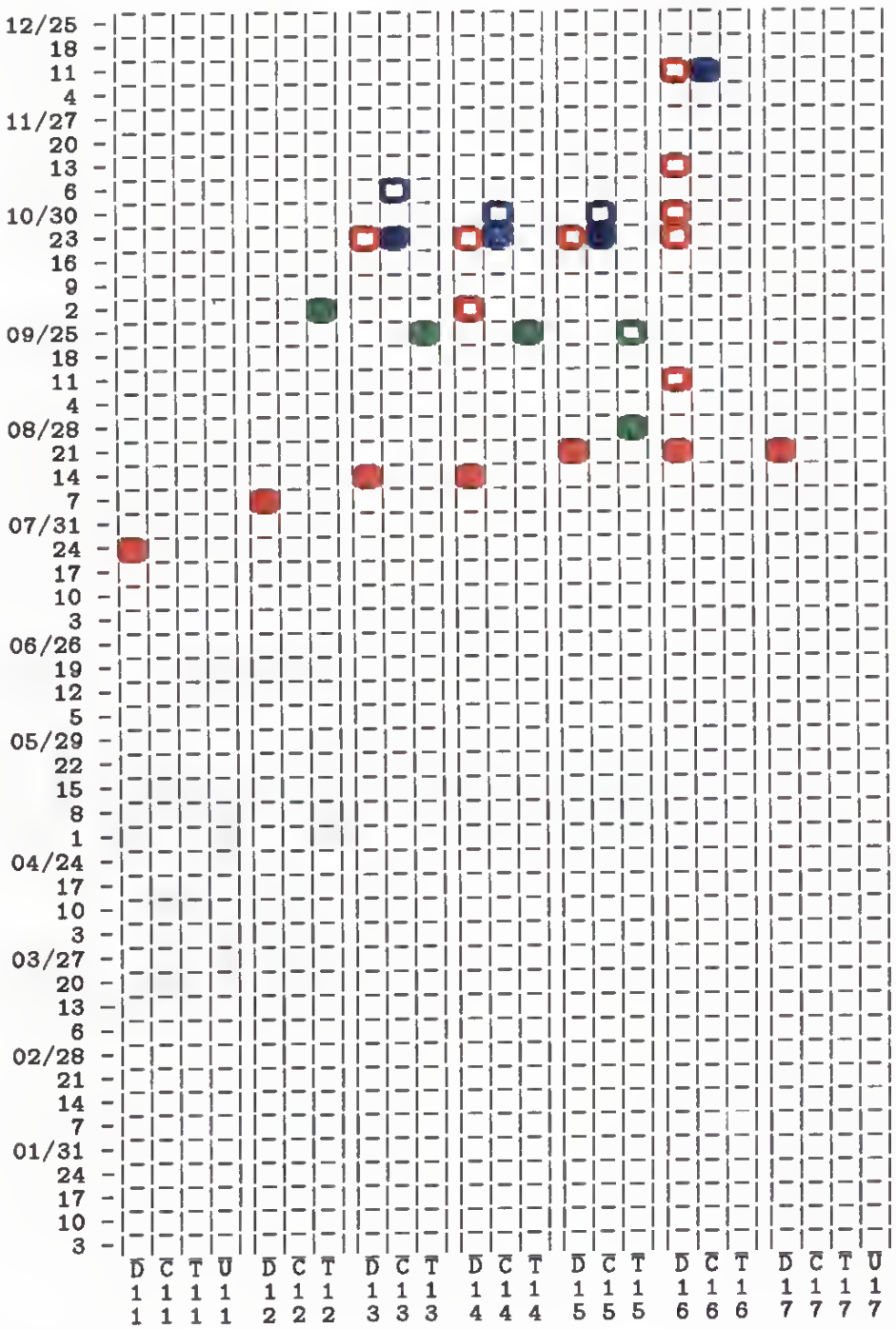
SPM FOR DOCUMENTS PRODUCED IN 1988



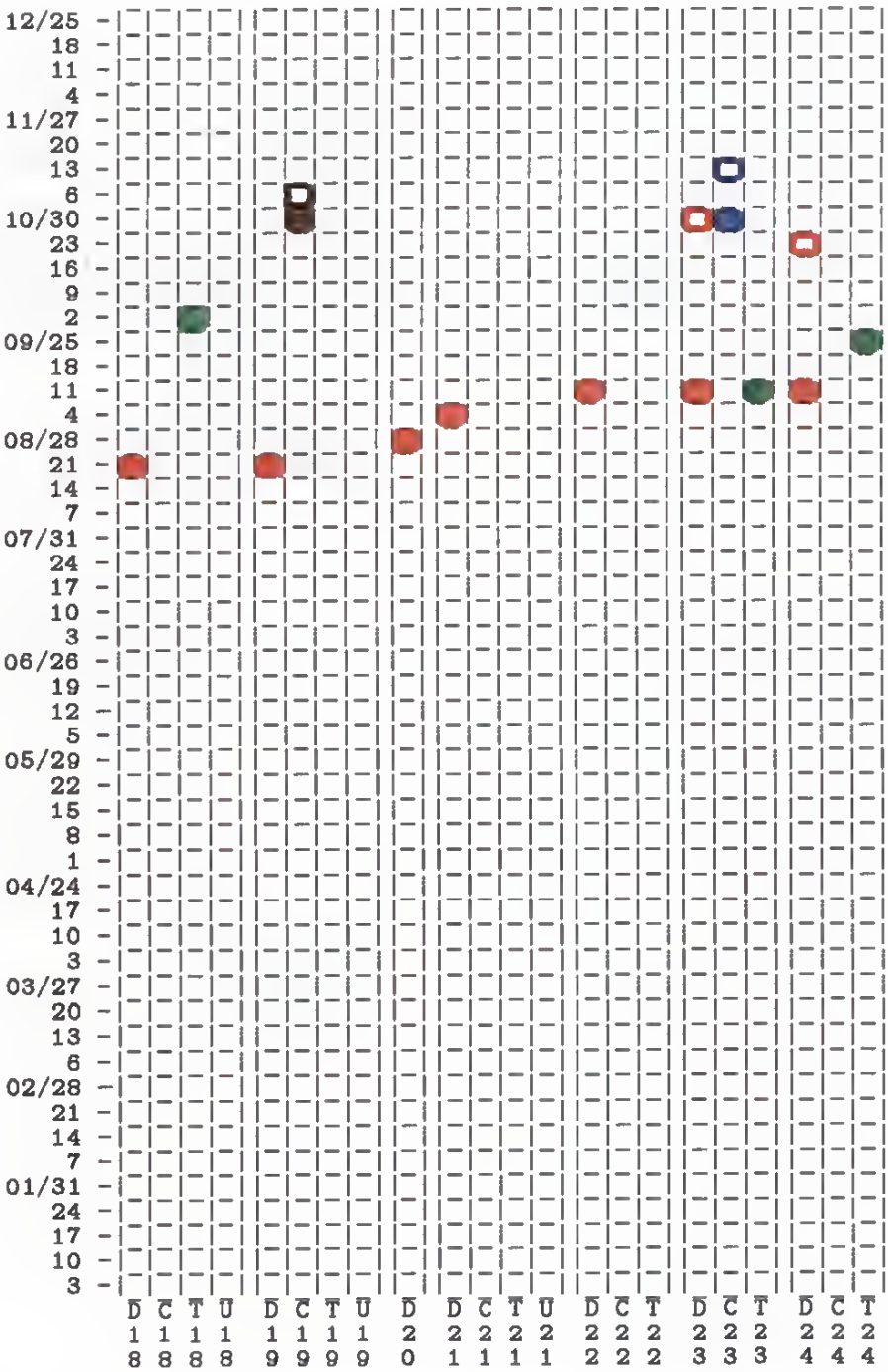
SPM FOR DOCUMENTS PRODUCED IN 1988 (cont)



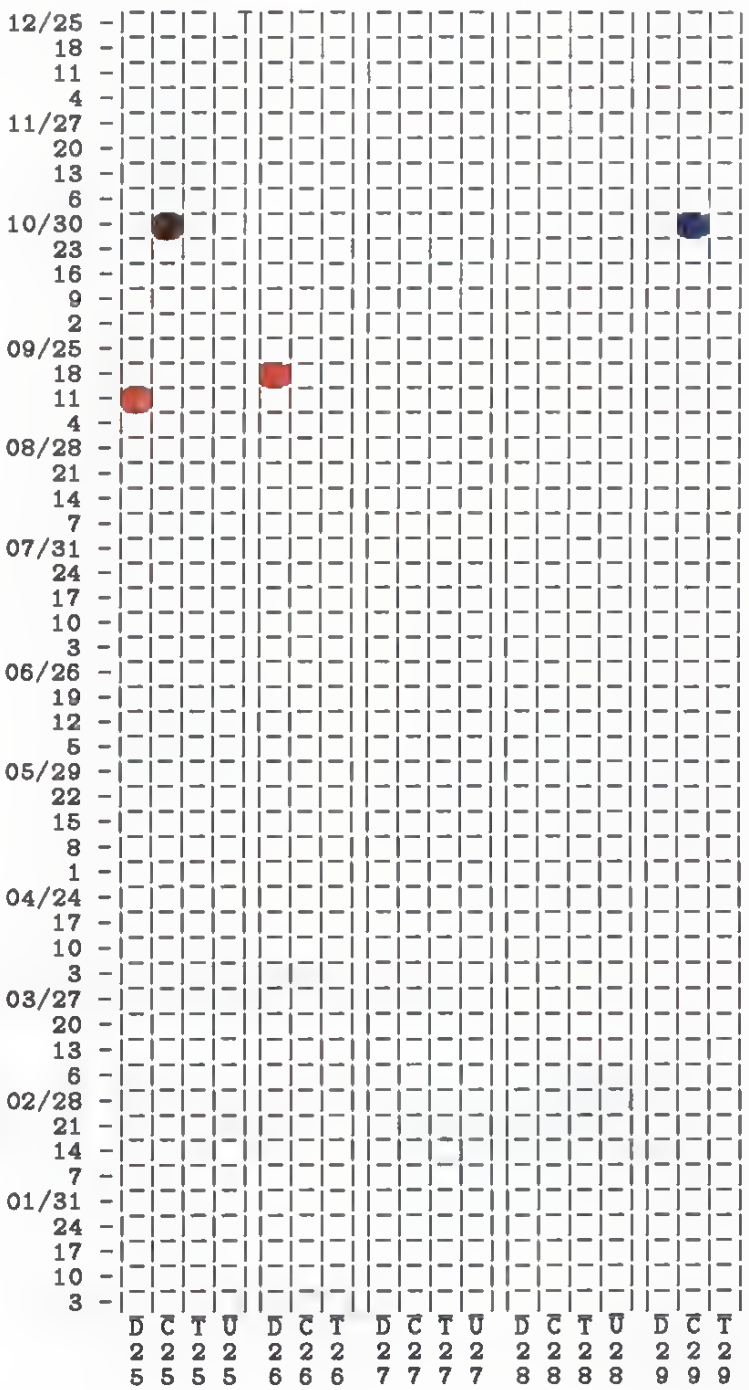
SPM FOR DOCUMENTS PRODUCED IN 1988 (cont)



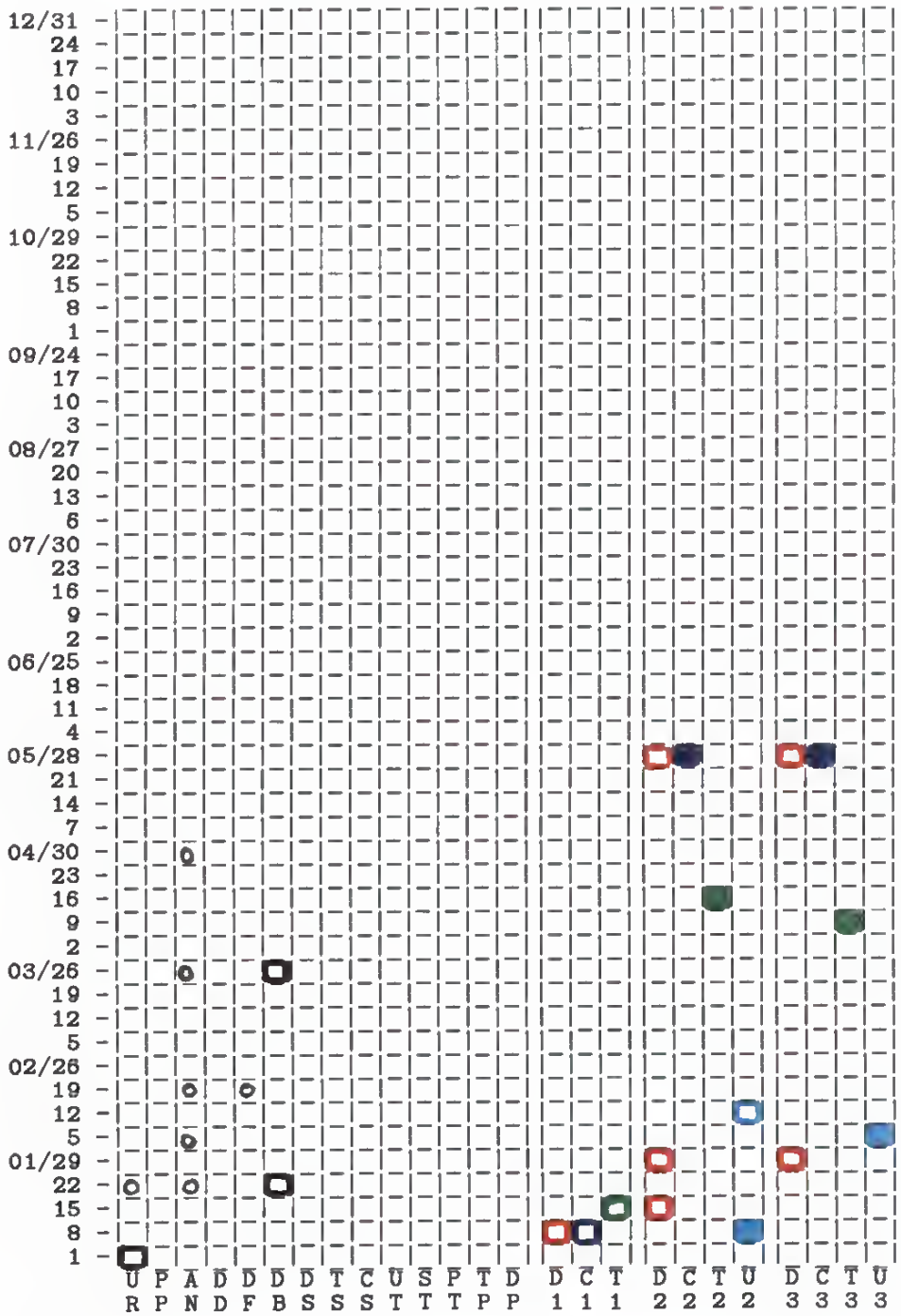
SPM FOR DOCUMENTS PRODUCED IN 1988 (cont)



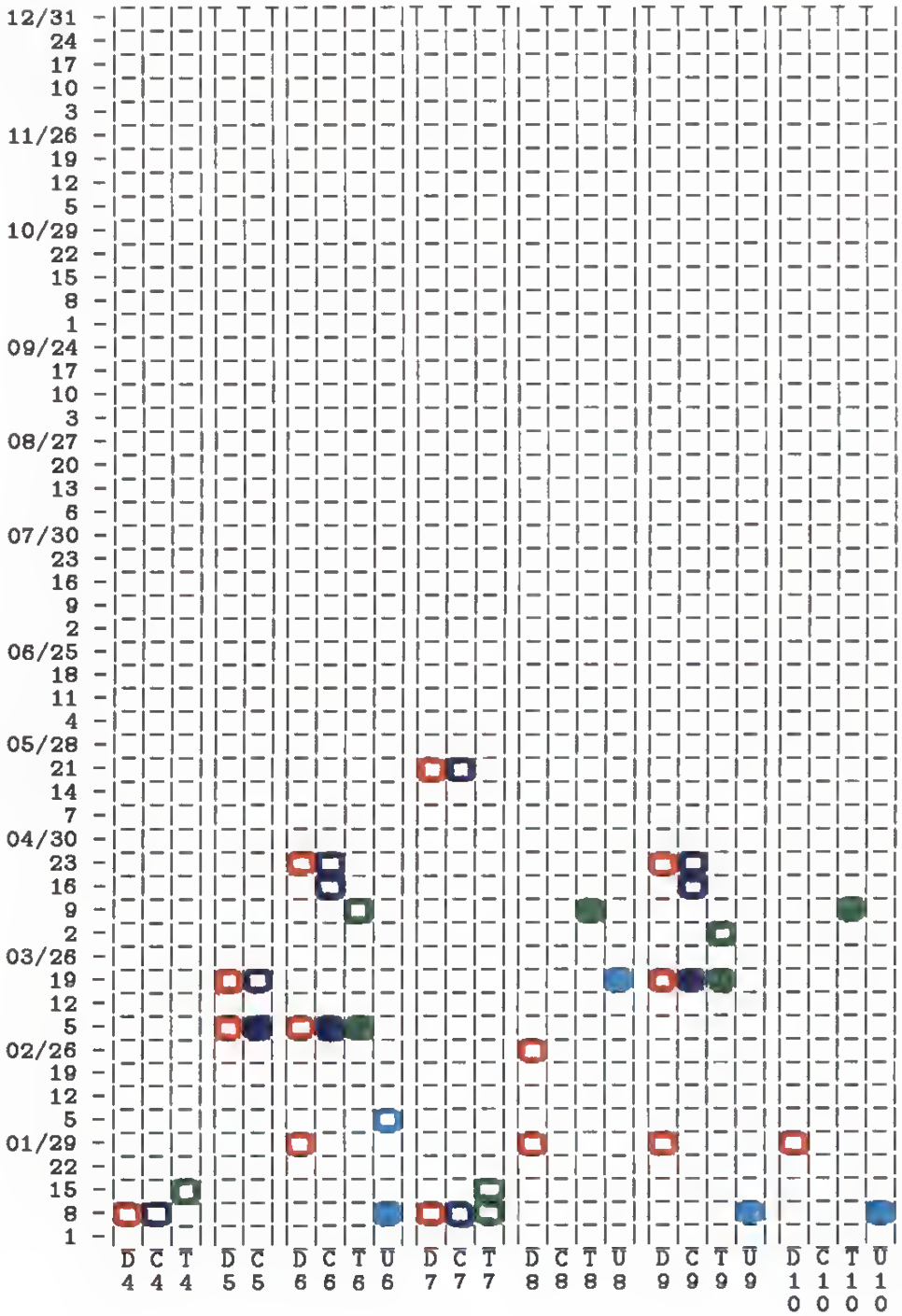
SPM FOR DOCUMENTS PRODUCED IN 1988 (cont)



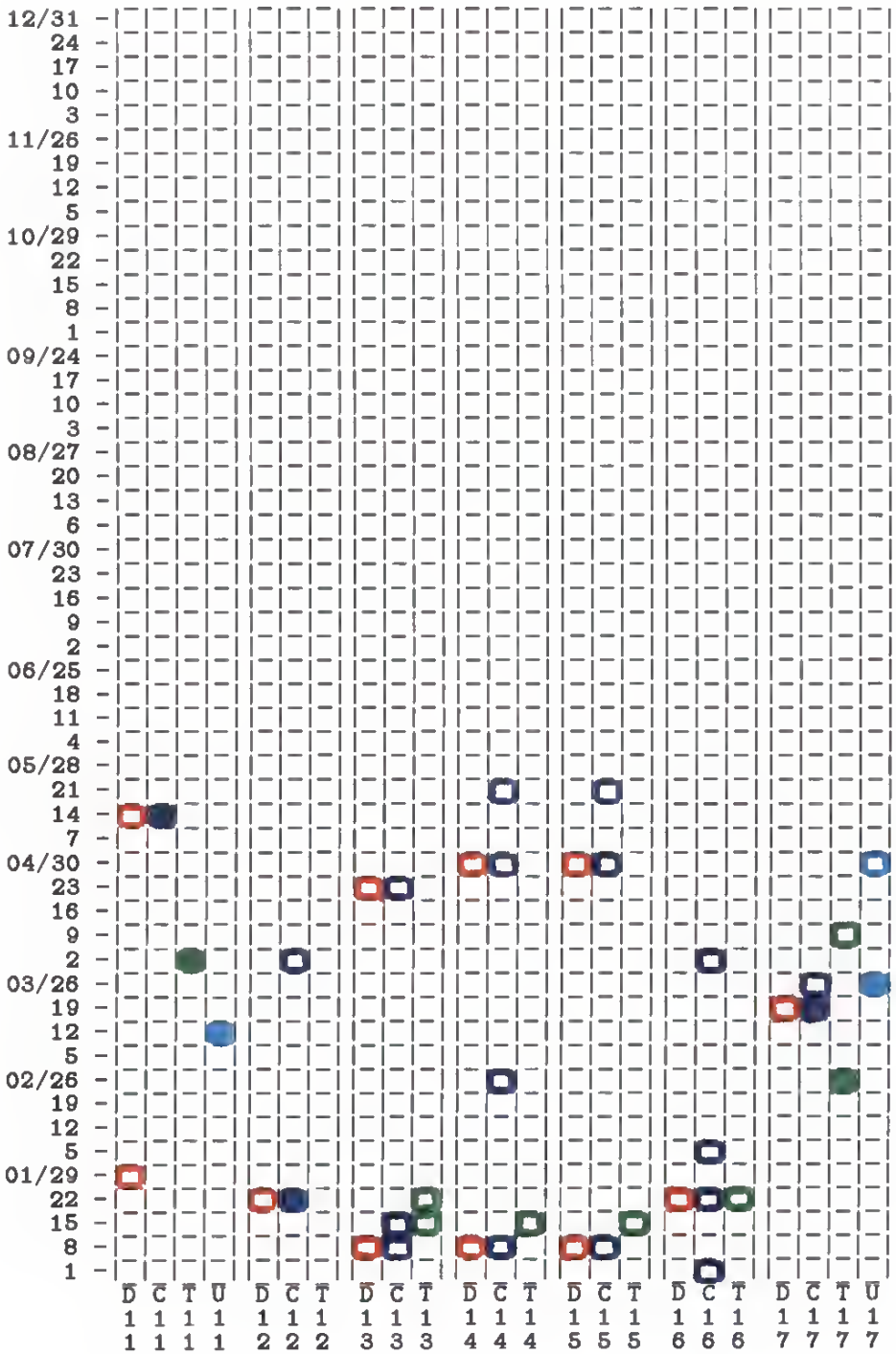
SPM FOR DOCUMENTS PRODUCED IN 1989



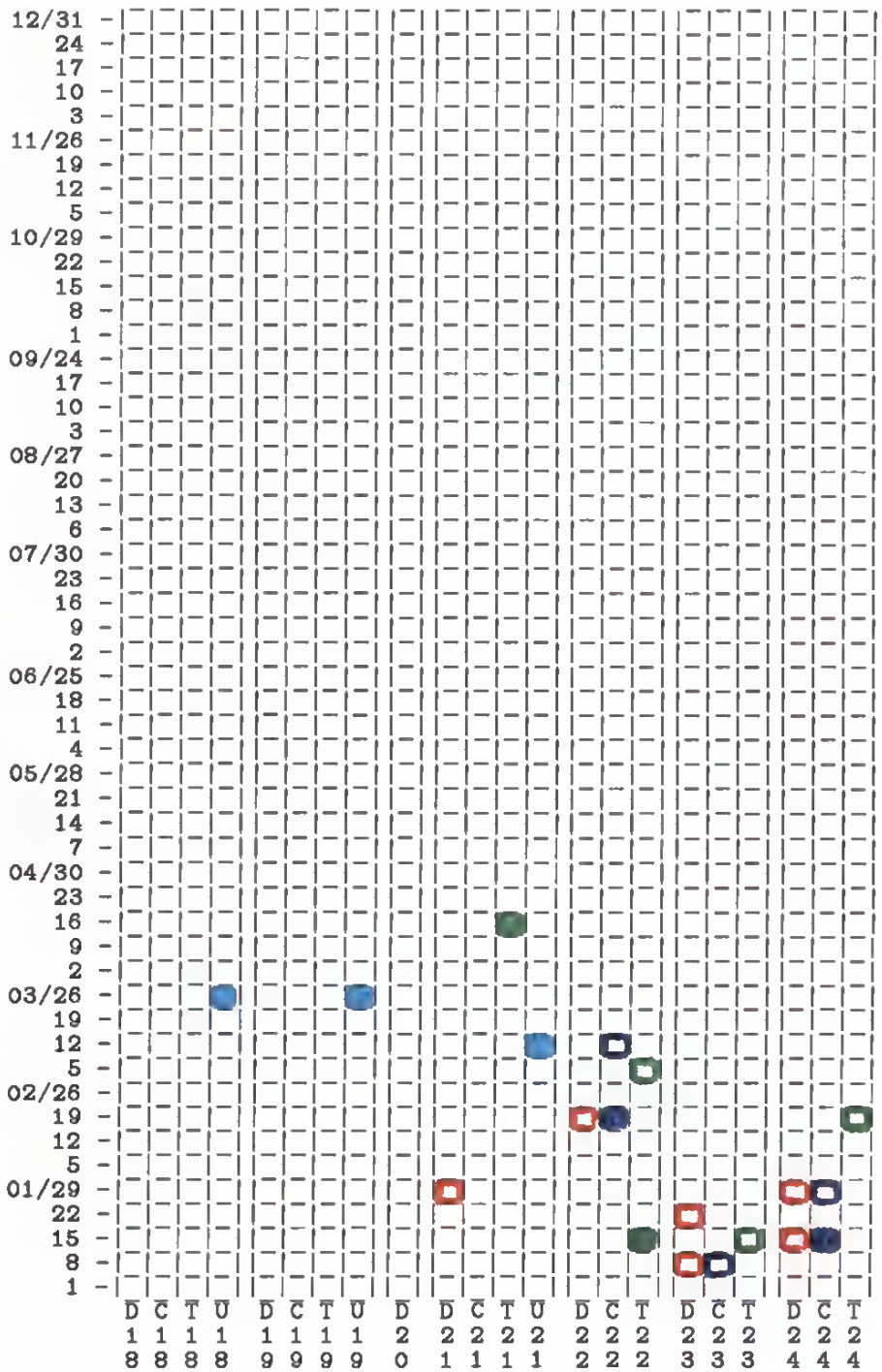
SPM FOR DOCUMENTS PRODUCED IN 1989 (cont)



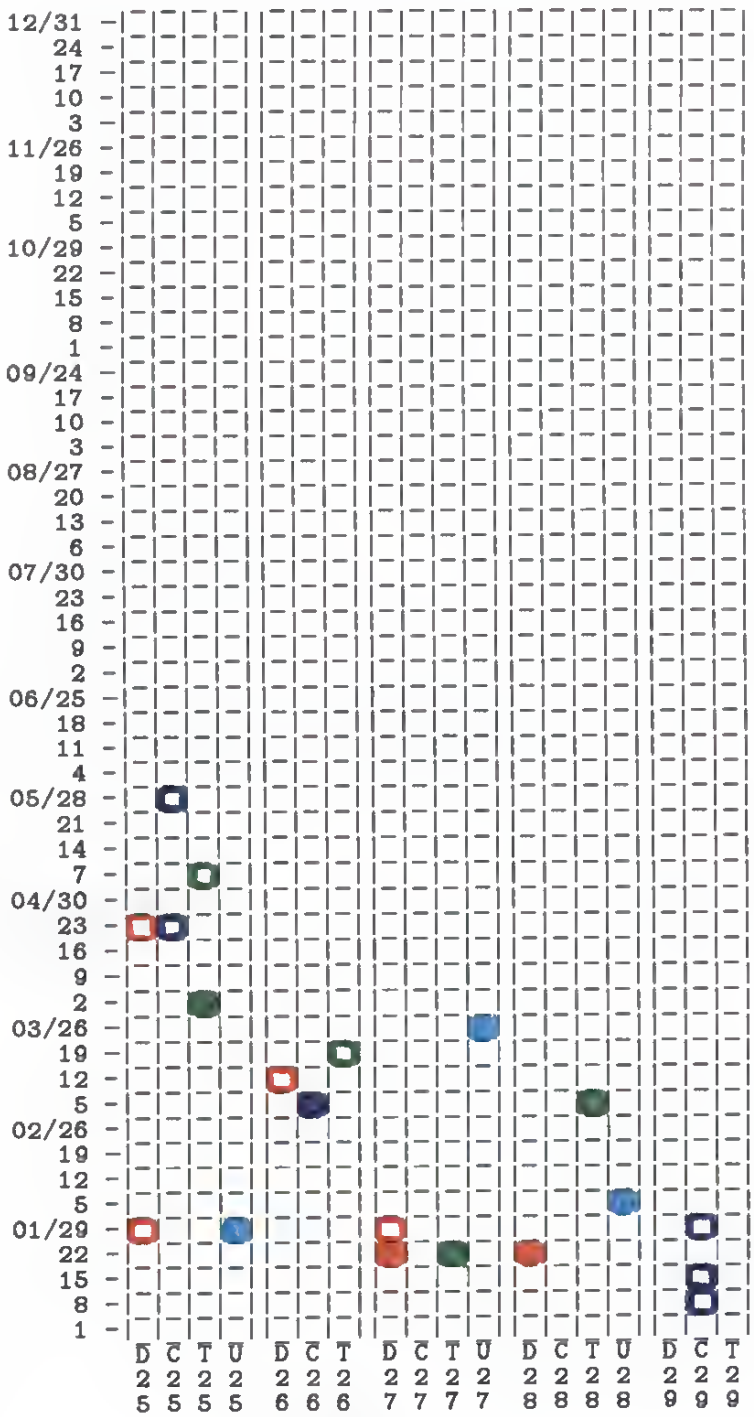
SPM FOR DOCUMENTS PRODUCED IN 1989 (cont)



SPM FOR DOCUMENTS PRODUCED IN 1989 (cont)



SPM FOR DOCUMENTS PRODUCED IN 1989 (cont)



USING THE SOFTWARE PROCESS MODEL TO ANALYZE A SOFTWARE PROJECT

by

I. SUE RANFT

B.S., Ohio State University, 1983

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

Software development has grown rapidly over the years. Models of the software development process have been developed to provide a means of controlling and visualizing software projects. In 1987 the Software Process Model (SPM) was introduced. The SPM models software development by modeling the evolution of the full set of documents produced in a software process.

This paper evaluates the usefulness of the Software Process Model as a management tool by applying it to a software project. This paper describes the SPM and the software project to which it was applied. It also presents information which was generated by reviewing the project from the viewpoint of the SPM and demonstrates the SPM's usefulness in managing a software project.