Chapter 1

Software Tools

In order to carry out the case study, I developed two software tools: the first to automate the PSP and the second to track errors found in the PSP projects. Both tools were written using the Progress 4GL/RDBMS [1], version 6.3F01, running on SCO Unix, Release 3.2v4.2, using a character based interface.

1.1 PSP Tool

1.1.1 Requirements

Obviously, the main requirement for this package was the ability to reliably automate all the PSP calculations so that I could check the original PSP data for errors. Since for this purpose I was the only user, a clear user interface and flexible functionality weren't very important. However, I also wanted to go beyond what was necessary for simple data checking and explore some ideas about a fully integrated set of PSP tools.

When I first learned the PSP, I almost immediately began creating various small tools to help me follow the PSP processes more efficiently. Some of these, such as a Java LOC counter were assignments in the PSP course itself. I created others, such as a size estimation applet, simply because I got tired of doing various processes by hand. Even though these tools relieved me of a lot of tedious work, it still seemed as though there was a tremendous amount of overhead involved in using the PSP to do actual software development. I would have a stack of PSP forms (with the correct one never on top) to the left of my keyboard, a stack of prior projects beyond that, “A Discipline for Software Engineering” (never open to the right page) for process scripts and form instructions under my elbow, and to the right all the papers needed for whatever software project I was working on. Of course, my pencil, eraser, and calculator always seemed to be *under* one of these papers whenever I needed them! On the computer, I had to continuously invoke various tools, provide input, and write down results on one of the PSP forms. All this led me to form the requirement that the new tool should completely eliminate any need for paper, including the need for the textbook for process scripts and form instructions. (Of course, the book is still needed to *learn* the PSP, but not as a day-to-day reference guide.) It should seamlessly combine all tools in such a way that the user never has to consciously invoke a tool or transfer data from one tool to another.

Closely related to this was the requirement that the new tool should provide as much guidance as possible in following process scripts. The user should not have to remember the order of the phases, the order of the forms within each phase, or which fields on a particular form are required for a particular phase. The tools should provide context-sensitive help screens and help messages. The computer should also calculate all possible values. It should provide derived values, such as *LOC per hour, Actual*, to the user in display rather than update mode. For other values, such as *Time in Phase, Total, Plan*, it should at least give the user a default number to override.

Since PSP results aren’t any better than the data that goes into them, another requirement was that the tool should do as much collection of primary data (time, size, and defects) as possible. Data, such as defects, should be easy to collect so that a user is less likely to have an incomplete record of his or her work. The system should also collect data automatically, whenever possible, so that a user is less likely to have an inaccurate record of his or her work. For example, the user should not have to fill in time log entries or tell the tool how many LOC are in a certain program. However, the user should be able to override or correct any automatically collected data.

One of the most aggravating things about doing PSP by hand is the transfer of data between projects. Size and time estimation require data from as many prior projects as possible. The user must either maintain a spreadsheet of this data, or continually leaf through a stack of former projects, looking for the relevant fields. Even more data is required from the most recently completed project, in both the planning and postmortem phases. The user must find the PSP forms for this project, and then reference various values at least 45 times (for PSP2), not counting size and time estimation. Even worse is the situation where a programmer discovers an error in PSP data for an older project or decides to declare a prior project an outlier. Recalculating to date values through prior projects up to the current project is no fun at all! Therefore, a requirement for the new tool was the ability to do inter-project management. The tool should allow the user to categorize each project by PSP process, programming language, and process type; as well as allowing certain projects to be marked as outliers. Other than the PSP process, the user should be able to change any of these characteristics even after the project is complete. The tool should have the ability to recalculate all PSP data for all prior projects at any time. Furthermore, when the user is working on a new project, the tool should automatically reference relevant prior projects to provide all appropriate "carry forward" values.

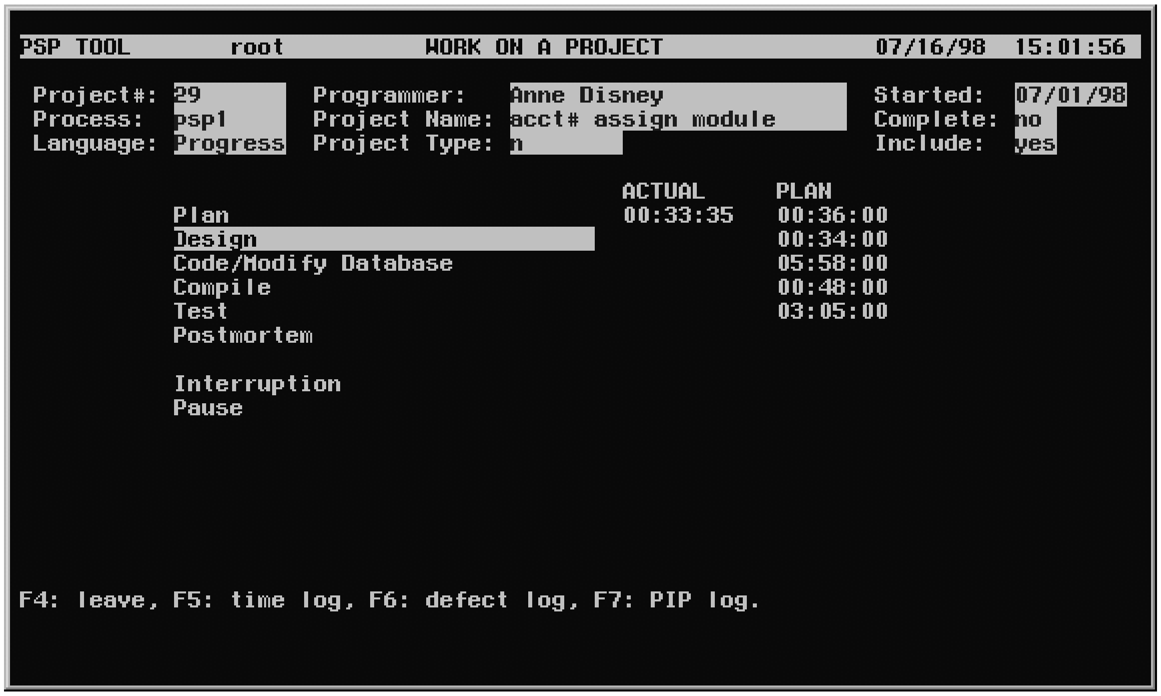
Since reports make collected PSP data even more useful, the ability to do at least the reports outlined in "A Discipline for Software Engineering" was another requirement.

A final requirement was that the tool should be as flexible as possible. Since Humphrey intended that developers should modify the PSP for their own needs, users should be able to do such things as define new processes, change existing process definitions, maintain individual coding standards, or add new defect types. Users should also be able to customize the system by such actions as modifying help screens, adding or modifying table entries for programming languages, or changing menu option descriptions.

1.1.2 Operational Scenarios

1.1.2.1 Scenario One: Starting a Project Using PSP1

A developer decides to start work on a new software project using PSP1. She goes to the main PSP menu and selects “Work on a Project”. The menu disappears and the header for the main project screen comes up. This header can be seen on the top part of the screen shown in Figure 1.1. The system prompts her for a project number. Since she is starting a new project, she follows the instructions in the help message and hits return on the blank field.



```
PSP TOOL        root          WORK ON A PROJECT            07/16/98  15:01:56

 Project#:  29        Programmer:    Anne Disney              Started:   07/01/98
 Process:   psp1      Project Name:  acct# assign module      Complete:  no
 Language:  Progress  Project Type:  n                        Include:   yes

                                                ACTUAL     PLAN
            Plan                                00:33:35   00:36:00
            Design                                         00:34:00
            Code/Modify Database                           05:58:00
            Compile                                        00:48:00
            Test                                           03:05:00
            Postmortem

            Interruption
            Pause






F4: leave, F5: time log, F6: defect log, F7: PIP log.
```

Figure 1.1: *PSP Tool: Main Project Screen*

The system looks in her personal profile for the next project number to assign, verifies that it hasn’t already been used, and assigns it to the new project, displaying “29” on the screen. It then prompts her for the other data elements needed for project management: process, programming language, programmer, project name, project type, date started, and outlier status. Based on system information and her personal profile, the system provides defaults for process, programming language, programmer, date started, and outlier status. The developer accepts these values and fills in the project name and assigns the project a type of “N” for new. She then presses F1.

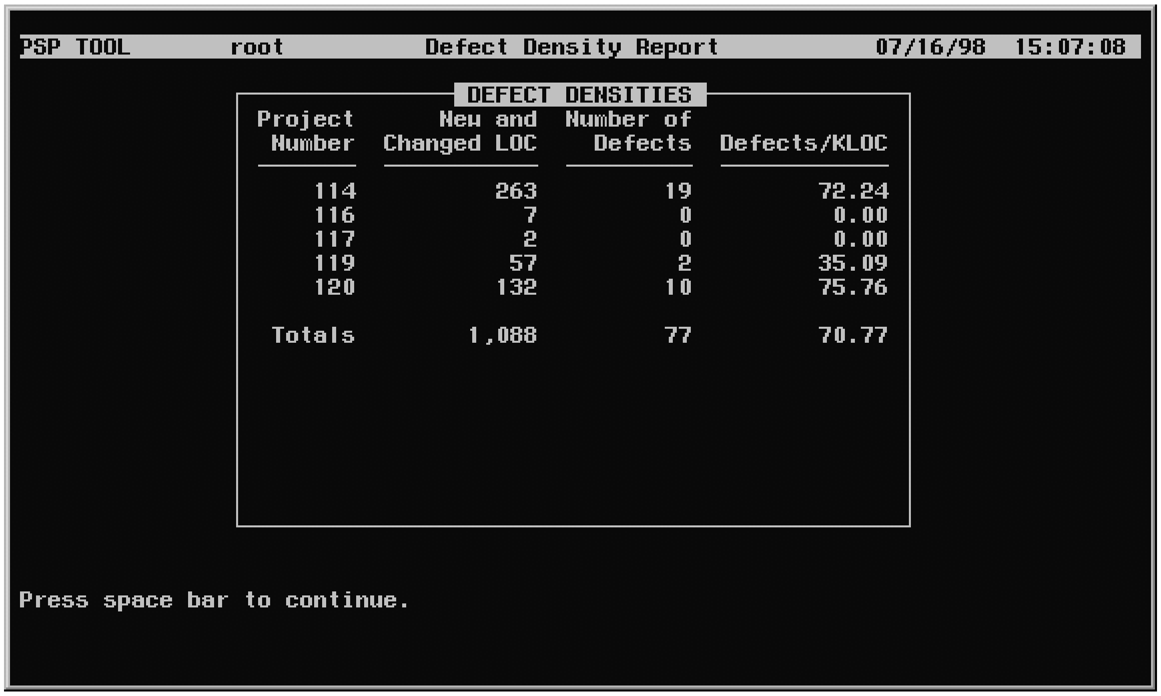
The system sees from the header data that she will be using PSP1. It finds the database record previously created to define process “PSP1”, builds a list of the phases that are involved, and displays their names on the screen. There are two additional columns for *Planned* time in phase and *Actual* time in phase. The developer knows that although she can arrow up and down over the phase list and select any one, the first phase listed is the phase she should use first if she plans to follow the standard process script. Therefore, she presses return on “Plan”. Immediately an asterisk appears by “Plan” on the screen, showing that this is the current phase. Under the *Actual* Time in Phase column, the amount of time spent in Planning is incremented second by second.

Immediately, a box appears titled, “Problem Description”. She types in “Client LYX requests WP interface to dictation sequences.” and presses F1. The box disappears and a new box comes up titled, “Requirements Statement”. She types in a number which references the hardcopy requirements document generated in a planning meeting the previous week, and presses F1. A text editor then fills the screen with a document already named, created, and pulled up for her, titled “Produce a Conceptual Design”. She takes about half an hour to do this work, and exits the editor. A new box appears titled “Program List”, which prompts her for the code location and the names of the programs she will be adding or modifying. The code location defaults to the one stored in her personal profile, so she just types in the name of the one program she will be modifying and the names of four new ones. After she presses F1, the system automatically counts and displays the number of lines in each program (zero for the new ones). It also makes backup copies so that after development it can determine the number of lines that were added, modified, or deleted.

Then the “Size Estimating Template” box appears, which prompts the developer for all the fields in the top part of the standard Size Estimating Template. After she fills them in and presses F1, another box appears, showing the results of all calculations done from the lower part of the standard Size Estimating Template. There is a message in the box informing her that calculations have been done, "Using projects from current language/process/type." The "Project Plan Summary" box appears, showing default values for *Program Size (LOC), Plan*: base, deleted, modified, added, reused, and total new reused. She glances at them and presses F1. Then the time estimation box appears, with an estimate for total time and a message "From regression calculation on estimated object LOC and actual hours." She accepts the default value of 661 minutes and presses F1. The system uses to date percentages from her most recent similar project to distribute the time across the phases, and again presents her with default values. She presses F1 again and is returned to the main project screen. The screen now displays the planned values for each phase, and "00:33:35" under *Actual* for the phase "Plan", as shown in Figure 1.1. She moves the cursor down one line to "Design". Invisibly, the system creates a time log entry for the planning phase.

1.1.2.2 Scenario Two: Defect Density Report

A software engineer has been conscientiously using the PSP for about six months, and wonders if the quality of his work on new projects is improving or not. He goes to the main PSP menu and selects "Reports". The system displays the Reports Menu and he selects "Defect Densities". The system hides the Reports Menu and displays the driver screen for the Defect Density Report. The screen prompts him for process, project type, language, date range and display type. Using his default values stored in database file, it fills in "PSP2" for process, "N" for project type (new projects), "Progress" for language, 06/12/98 and 09/04/98 for date range (first and last project completed of type "N" using PSP2), and "D" for display type. He doesn't need to change anything, and presses F1 to proceed. Instantly, a box appears with a line for each of the 25 new projects he completed in Progress using PSP2 during the selected date range. For each project the system displays project number, new and changed LOC, number of defects, and defects per KLOC. The last line shows total values and the average number of defects per KLOC, as shown in Figure 1.2. He is happy to see that the later projects do appear to have fewer defects.



```
PSP TOOL        root           Defect Density Report              07/16/98  15:07:08

                                  DEFECT DENSITIES
                    Project      New and  Number of
                     Number  Changed LOC    Defects   Defects/KLOC

                        114          263         19          72.24
                        116            7          0           0.00
                        117            2          0           0.00
                        119           57          2          35.09
                        120          132         10          75.76

                     Totals        1,088         77          70.77

Press space bar to continue.
```

Figure 1.2: *PSP Tool: Last Page of Multi-Page Defect Density Report*

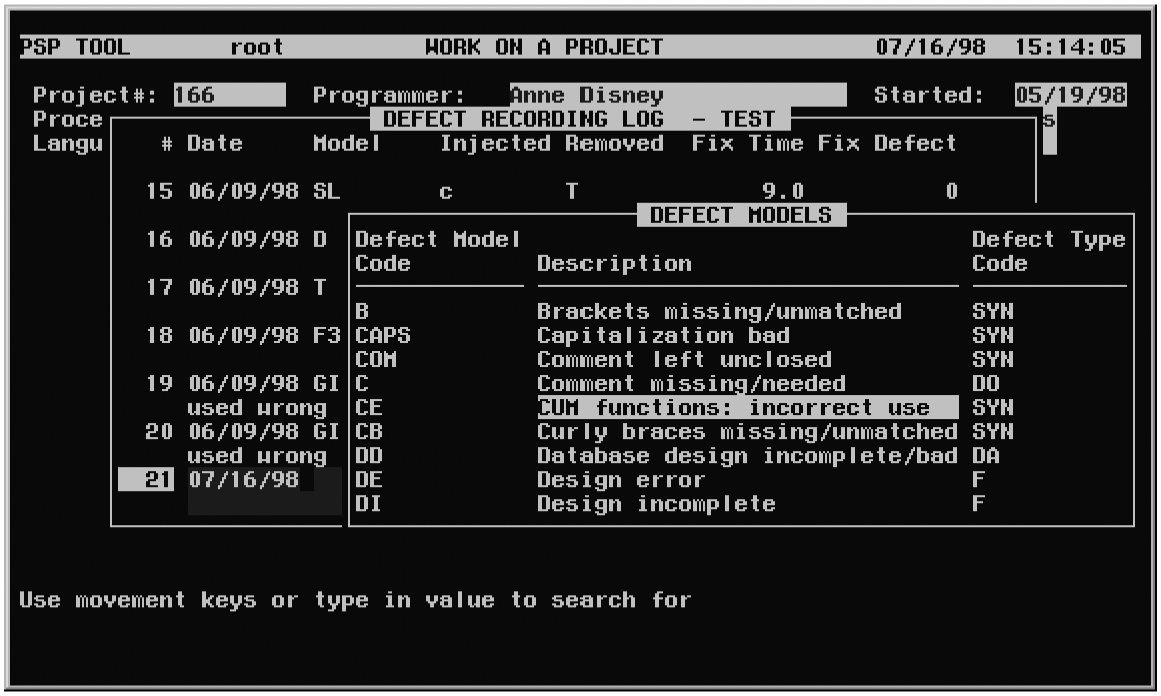
1.1.2.3 Scenario Three: Adding a Defect Model

Note: In the PSP tool, *defect types* are used as defined in “A Discipline for Software Engineering”, and refer to general defect categories such as Syntax, System, Environment, and Data. I have added a new classification, *Defect models*, which refer to specific defects within a defect type, such as “Missing semicolon” or “Confusing field label”. There are three reasons for this setup: 1) The user can record in the Defect Recording Log exactly what went wrong. This eliminates the mental overhead of deciding which defect type to use and cuts down on typing in explanatory comments. 2) It provides for a more consistent classification of defects. 3) It allows for finer grained defect reporting without extra work and without sacrificing any of the functionality required to produce the standard PSP defect reports using defect type.

A programmer notices that when recording defects he is continually using the defect model code “GSYN” for “General Syntax Error” and then using the comment line to explain “CUM function set error”. He decides that he wants to add a new defect model to the system to make his defect recording process faster. He goes to the main menu and selects the “Maintain Tables” menu option. A new menu appears. He then selects the “Defect Models” menu option. The menu disappears, and the Defect Model Code Maintenance screen appears.

Following the instructions in the help message, he decides on a new code “CE” and types it in. He then enters a description “CUM functions: incorrect use” and moves to the next field which is “Defect Type Code”. He doesn’t know what the defect type codes are, so he presses F6 and a list of all defect types appears. Using the down arrow key, he moves through the list until “Syntax” is highlighted and presses return. The defect type code “SYN” is filled in automatically, and the addition of a new defect model is complete.

The next time he is testing, he finds a defect caused by his incorrect use of a CUM function. He opens a new defect. The system numbers the new defect, fills in the date automatically, and places his cursor on the “Model” code field. He doesn’t remember the new code for CUM errors, so he presses F6 and is presented with a list of all defect models. He types in “CUM” and presses return. “CUM functions: used incorrectly” appears under his cursor, as shown in Figure 1.3, and he presses return again. “CE” is automatically placed in the defect model code field. No comment is necessary.



```
PSP TOOL        root            WORK ON A PROJECT               07/16/98  15:14:05

 Project#: 166       Programmer:   Anne Disney               Started:  05/19/98
 Proce ┌──────────────DEFECT RECORDING LOG  - TEST ──────────────────┐  s
 Langu │   # Date     Model    Injected Removed  Fix Time Fix Defect  │
       │                                                             │
       │  15 06/09/98 SL       c        T          9.0          0    │
       │                 ┌──────────────────DEFECT MODELS──────────────────────┐
       │  16 06/09/98 D  │Defect Model                              Defect Type│
       │                 │Code          Description                 Code       │
       │  17 06/09/98 T  │────────────  ──────────────────────────  ───────────│
       │                 │B             Brackets missing/unmatched  SYN        │
       │  18 06/09/98 F3 │CAPS          Capitalization bad          SYN        │
       │                 │COM           Comment left unclosed       SYN        │
       │  19 06/09/98 GI │C             Comment missing/needed      DO         │
       │     used wrong  │CE            CUM functions: incorrect use SYN       │
       │  20 06/09/98 GI │CB            Curly braces missing/unmatched SYN     │
       │     used wrong  │DD            Database design incomplete/bad DA      │
       │  21 07/16/98    │DE            Design error                F          │
       │                 │DI            Design incomplete           F          │
       └─────────────────└─────────────────────────────────────────────────────┘


Use movement keys or type in value to search for
```

Figure 1.3: *PSP Tool: Browsing for a defect model while adding a new entry to the Defect Recording Log*

1.1.2.4 Scenario Four: Fixing a Completed Project

A software developer has an extra half hour late one Friday afternoon. She doesn't want to start anything new, so she starts going through the stack of papers that tends to accumulate on one of the back corners of her desk. She finds a slip of paper that says: "McNall lab interface, 3 hours design, 03/22". She remembers that one Sunday back in March she did some work at home and never recorded it in her PSP database.

She turns to her screen and selects "Work on a Project" from her main PSP tool menu. The system displays the main project screen and prompts her for "Project#". Of course she can't remember the project number, so she presses F6 and selects the correct project from the list that appears. When the header fields appear, she changes the project status to "Incomplete" and presses F1. Following the help message on the bottom of the screen, she presses F5 and the project time log appears. She verifies that no time was recorded for March 22. Again following the help message, she presses F9 and creates a time log entry for the three hours of design work. Then she presses F4 and returns to the main project screen. She selects the postmortem phase. All the recalculations take about 1 second. She then presses the space bar as 6 boxes showing postmortem data appear. The system automatically marks the project as complete, and returns her to the main project screen, where she presses F4 twice. When the system returns her to the main PSP menu, she selects "Recalculate To-Date Values". A message appears "Working on project number 1." The project number quickly changes as the system recalculates the to date values for all appropriate projects using the new time measures for the changed project. 10 seconds later, the system is completely updated.

1.1.3 Structure

The PSP tool consists of about 80 programs and include files (subroutines) and a Progress database. The programs and include files add up to approximately 7000 lines of code. The database has 24 files (tables), as shown in Figure 1.4, containing 237 fields (columns). Each field definition includes a default validation expression, help message, display format, field description, and screen label.

I designed the system so that certain database records (rows) must be pre-loaded for the system to function. These records all belong to files whose names begin with the prefix "s-", and include menus, menu options, and links between certain fields and programs providing "browses" of user-defined acceptable values. Unless the user modifies the system through further programming, nothing should be added to or deleted from this record set. The only thing that may be modified are the names of menus and the names and numbers of menu options.

Unless a user wants to define his or her own processes completely from scratch, all files beginning with the prefix "c-" should also have a basic set of records pre-loaded before

Figure 1.4: Database Files for PSP Tool

File Name	Description
s-browse	Ties field names to code look-up browses
s-menu	Numbers and descriptions of menus
s-option	Numbers and descriptions of menu options
c-control	Default values for individual users
c-dmodel	Defect models (general defect categories)
c-dtype	Defect types (specific defect categories)
c-form	PSP forms
c-help	Help screens by process, phase, form
c-language	Programming languages
c-otype	Object types (for size estimation)
c-phase	PSP phases
c-process	Holds ordered sets of phases for processes
c-prophase	Ties phases to processes
c-worktype	Project types
defect	Defect Recording Log entries
lesson	Lessons learned/notes
PIP	Process Improvement Proposals
prob-desc	Problem descriptions
program-set	Programs created or modified during development
project	Projects
requirements	Requirements
size-template	Size Estimating Template data
test-report	Test Report Template entries
time-log	Time Recording Log entries

the user tries to start work. These records will define the standard PSP processes, forms, defect types, phases, etc. They will also provide a starting list of programming languages, object types, and defect types. Full facilities are provided to the user to modify these files and to add new records.

All the remaining files are records of the user's work and PSP data. The central file to this group is "Project". Records in the other files have a one-to-one or many-to-one relationship with records in the "Project" file. For example, there is one "requirements" record per project record and many "time-log" records.

I tried to design the system in such a way that programs rarely call other programs. Instead, database records determine which program to run next. Each menu option record contains a field for "program to run". After writing a simple report program, a user can add it to their system simply by creating a new menu option record using the maintenance function already written. The records used to define PSP forms also contain a "program to run" field. To define a new process, a user builds a list of the phases that the process will contain, then indicates the sequence of forms to use in each phase. This makes it easier for someone to write new programs and add them into the system, since he or she doesn't have to understand complicated program interactions.

1.1.4 Areas for Improvement

As with most applications, the PSP tool has several areas that could use more work. The main deficiency is that the standard PSP has not been completely implemented. The only processes that are truly complete are PSP0 and PSP0.1. PSP1 is missing an entry mechanism for the Test Report Template. This is because I was never able to devise a smooth way of switching between this form and the Defect Recording Log. PSP1.1 does not include the Task Planning Template and the Schedule Planning Template. The Design and Code Review Checklists are not available for PSP2. Finally, PSP3 has not been implemented at all.

There are also two problem areas. The first is measuring program size. At least for programs written in the Progress 4GL, this is not a simple matter of counting the number of lines in the program, since what I really want to measure is the number of statements used. This involves a tremendous amount of string parsing. I was able to implement this accurately in the Progress 4GL, but it takes a long time - about .1 seconds per line. Since this is tightly integrated with the rest of the PSP tool, it is still an improvement over other line counting methods, but the function should probably be rewritten in C++ or even a Unix script and called by the PSP tool. The second problem involves time recording. When a user moves too rapidly between phases, a time log entry is not recorded. I'm not sure if this is a problem with my code or the outcome of a combination of quirks in the Progress 4GL.

Of course, there are also areas that could be enhanced. It would be nice to add a table for editor types which would include pointers to programs to run to create, update, and access documents using editors other than vi! Then each user's personal profile could keep track of their preferred editor and automatically use it when needed. In another area, the *Actual* time in phase column on the main project screen should automatically refresh itself after the user manually changes the time log. Similarly, when a user changes any header data for a completed project, the system should automatically recalculate all to date values for the projects that follow it, rather than requiring the user to take a second step. Finally, the programs that provide maintenance of the defect types, defect models, phases, forms, etc., do not allow the user to delete entries. The user should be able to request a deletion, then the system should check all relevant records to see if the code has been used at all. If not, the system should delete the record, otherwise it should issue a warning message to the user.

1.2 Error Tracking Tool

1.2.1 Requirements

From the beginning, I viewed the error tracking tool as a special-purpose, single-user application; with the special purpose of recording errors found while analyzing PSP data, and the single user as myself. Therefore, the requirements were rather simple.

The primary requirement was that the tool should have an appropriate database structure for recording data about the PSP errors and should provide an entry screen that would allow me to record the errors quickly and accurately. If a field on the entry screen corresponded to a list of acceptable values, then the tool should allow me to enter only one of those values.

As I found errors, I wanted to keep track of 11 pieces of data for each one; such as person who injected, person who discovered, assignment, process, phase, form, etc. (This does not include three data elements that did not apply to every error: incorrect value, correct value, and comment lines). 9 of the 11 fields would contain codes that would represent items from lists of acceptable values. Some of the lists, such as severity levels, would contain a fixed set of values. But others, such as defect types, would probably grow as I worked through the case study. Therefore, another requirement was that the lists of acceptable values should have maintenance programs available for adding and modifying entries, and that each list should appear in selection mode after the user pressed F6 from the corresponding field on the error entry screen.

A final requirement was that the tool should allow me to analyze the error data after it had been entered. It should do this by creating various reports. Since these were to be special-purpose reports created for a single user, there was no need for fancy output formats or drivers with long lists of options.

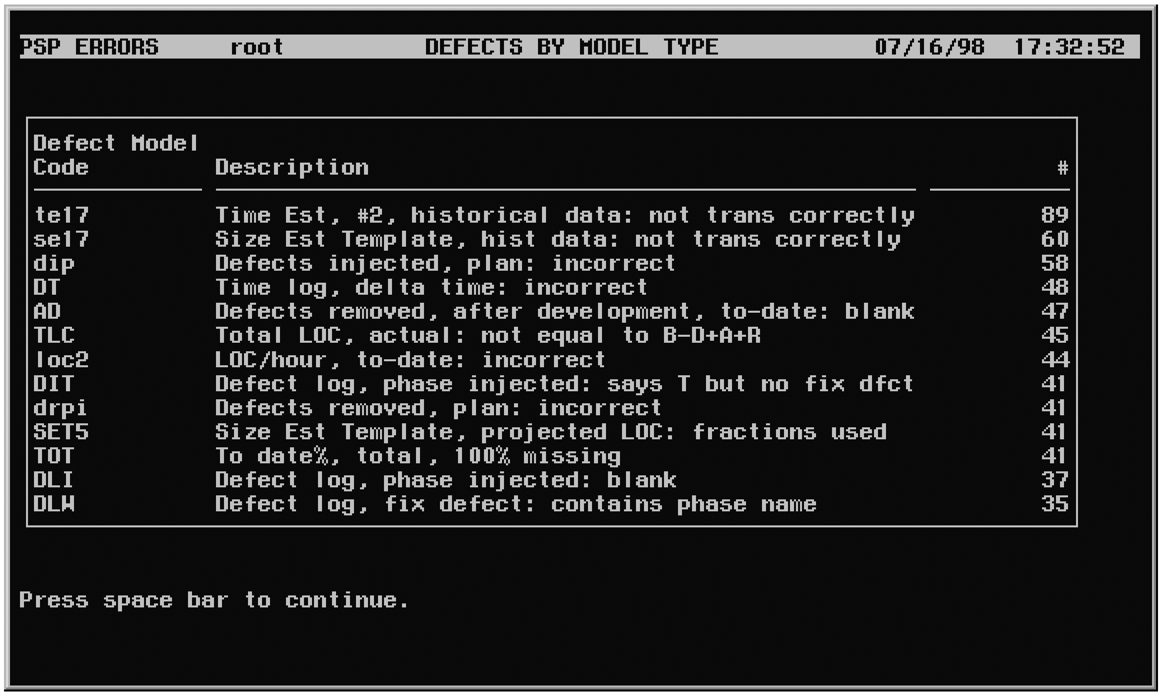
After implementing this tool and entering about half the projects, I realized that I needed to keep track of estimated LOC, actual LOC, and actual minutes as they were *originally* recorded on the student projects. This was necessary to allow me to efficiently duplicate time and size estimation and LOC calculation using incorrect data. Therefore, this became the main requirement for a small addition to the original application.

1.2.2 Structure

Figure 1.5: Database Files for Error Tracking Tool

File Name	Description
s-browse	Ties field names to code look-up browses
s-menu	Numbers and descriptions of menus
s-option	Numbers and descriptions of menu options
c-dmodel	Defect models (general defect categories)
c-dtype	Defect types (specific defect categories)
c-form	PSP forms
c-language	Programming languages
c-person	People (instructor, students, myself)
c-phase	PSP phases
c-process	PSP processes
c-severity	Defect severity levels (impact of errors on other data
c-worktype	Project types
defect	Errors found in student PSP data
hist	Summary of project data by author/assignment (Used to keep key data on-line so that when doing calculations by hand I didn't have to look through stacks of prior projects)

The error tracking tool consists of about 60 programs and include files (subroutines) and a Progress database. The programs and include files add up to approximately 2000 lines of code. The database has 14 files (tables), as shown in Figure 1.5, containing 49 fields (columns). Each field definition includes a default validation expression, help message, display format, field description, and screen label. Besides providing tracking of errors and historical data, it generates 12 reports. A sample report is shown in Figure 1.6.



```
PSP ERRORS      root          DEFECTS BY MODEL TYPE          07/16/98  17:32:52


 Defect Model
 Code            Description                                                  #

 te17            Time Est, #2, historical data: not trans correctly          89
 se17            Size Est Template, hist data: not trans correctly           60
 dip             Defects injected, plan: incorrect                           58
 DT              Time log, delta time: incorrect                             48
 AD              Defects removed, after development, to-date: blank          47
 TLC             Total LOC, actual: not equal to B-D+A+R                     45
 loc2            LOC/hour, to-date: incorrect                                44
 DIT             Defect log, phase injected: says T but no fix dfct          41
 drpi            Defects removed, plan: incorrect                            41
 SET5            Size Est Template, projected LOC: fractions used            41
 TOT             To date%, total, 100% missing                               41
 DLI             Defect log, phase injected: blank                           37
 DLW             Defect log, fix defect: contains phase name                 35


Press space bar to continue.
```

Figure 1.6: *Error Tracking Tool: Sample Report*

Bibliography

[1] Progress software. Information is available at: www.progress.com/core/develop.htm.