Estimating the Value of Inspections and Early Testing for Software Projects

A return-on-investment model is developed and applied to a typical software project to show the value of doing inspections and unit and module testing to reduce software defects.

by Louis A. Franz and Jonathan C. Shih

The software inspection process has become an important part of the software development cycle,[1,2,3,4] and has been used with varying levels of success within Hewlett-Packard.[4,5] One of the main reasons for its success is that detecting defects early has a big impact on reducing the cost of dealing with software defects later in the development cycle. One HP entity used metrics data from several software projects and an industry profit and loss model to show the high cost of finding and fixing defects late in the development cycle and during postrelease.[5]

This paper describes the methods we have used to integrate inspections and prerelease testing into the development of an information technology software project. The metrics collected and the tools we used to collect the metrics data on this project are described. Finally, we describe an approach to using the metrics data collected during inspections and testing to estimate the value (return on investment) of investing time and effort in early defect detection activities.

Background

The sales and inventory tracking (SIT) project evolved from separate initiatives by several different groups in HP. These initiatives had different objectives, but all relied on elements of the same data: computer dealer sales and inventory levels of HP products. To simplify the collection, processing, and storage of this data, it was decided to create a centralized system to house the data. All the applications that would need to use this information could access the central SIT system database. Fig. 1 shows the four major modules that make up the SIT system and the major data stores accessed by the system. These modules will be referenced throughout this paper. Table I provides a summary of the major attributes of the system.

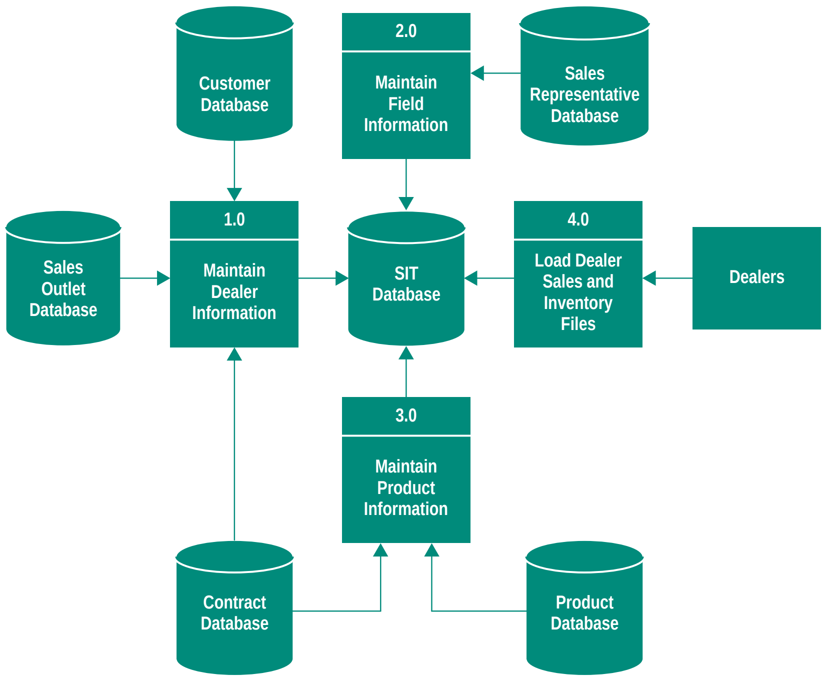

Customer Database
2.0
Maintain Field Information
Sales Representative Database
1.0
Maintain Dealer Information
Sales Outlet Database
SIT Database
4.0
Load Dealer Sales and Inventory Files
Dealers
3.0
Maintain Product Information
Contract Database
Product Database

Fig. 1. The major components of the sales and inventory tracking (SIT) system.

Table I
Summary of the Sales and Inventory Tracking System

System type	Batch
Hardware platform	HP 3000 Series 980 running the MPE/XL operating system
Language	COBOL
DBMS	HP AllBase/SQL (relational)
Data communications (used for the electronic transfer of data between the dealer and HP)	Electronic Data Interchange (EDI) ANSI standard transaction sets (file formats): 867 product transfer and resale report; 846 inventory inquiry and advice
Data access and manipulation tools	Cognos, PowerHouse (Quiz), Supertool, KSAM, and VPLUS
Code	25K total lines of source code (13.6 KNCSS)*

* KNCSS = Thousands of noncomment source statements

We used the traditional HP software life cycle for our development process. The basic steps of this process include:

- Investigation
- Design
- Construction and Testing
- Implementation and Release
- Postimplementation Review.

The inspection and prerelease testing discussed in this paper occurred during the design, construction, and testing phases.

Inspection Process and Inspection Metrics

The objectives of the inspection process are to find errors early in the development cycle, check for consistency with coding standards, and ensure supportability of the final code. During the course of conducting inspections for the SIT project, we modified the HP inspection process to meet our specific needs. Table II compares the recommended HP inspection process with our modified process.

Step 3 was changed to Issue and Question Logging because we found that inspectors often only had questions about the document under inspection, and authors tended to feel more comfortable with the term issue rather than defect. The term defect seemed to put the author on the defensive and severely limited the effectiveness of the inspection process.

Table II
Comparison of Inspection Processes

Step	Our Inspection Process	HP Inspection Process
0	Planning	Planning
1	Kickoff	Kickoff
2	Preparation	Preparation
3	Issue and Question Logging	Defect Logging
4	Cause Brainstorming	Cause Brainstorming
5	Question and Answer	Rework
6	Rework	Follow-up
7	Follow-up	

The Question and Answer step was added because inspectors often had questions or wanted to discuss specific issues during the Issue and Question Logging session. These questions defocused the inspection and caused the process to take longer.

In the Planning step the author and the moderator plan the inspection, including the inspection goal, the composition of the inspection team, and the meeting time. The Kickoff step is used for training new inspectors, assigning the roles to the inspection team members, distributing inspection materials, and emphasizing the main focus of the inspection process. During the Preparation step, inspectors independently go through the inspection materials and identify as many defects as possible. In the Cause Brainstorming step, inspection team members brainstorm ideas about what kind of global issues might have caused the defects and submit suggestions on how to resolve these issues. During the Rework step, the author addresses or fixes every issue that was logged during step 3. Finally, in the Follow-up step, the moderator works with the author to determine whether every issue was addressed or fixed.

Along with the modified process, a one-page inspection process overview was generated as the training reference material for the project team. This overview was a very convenient and useful guideline for the project team because it helped to remind the team what they were supposed to do for each inspection.

Deciding What to Inspect

Because of time and resource constraints, not all of the project's 13 source programs and 29 job streams could be inspected. The project team decided to use the risk assessment done for the master test plan, which uses the operational importance and complexity of a module as a basis for deciding which programs and job streams to inspect. The risk assessment used for the master test plan is described later. Fig. 2 shows the results of this selection process in terms of inspection coverage and relative level of complexity of the programs and job streams.

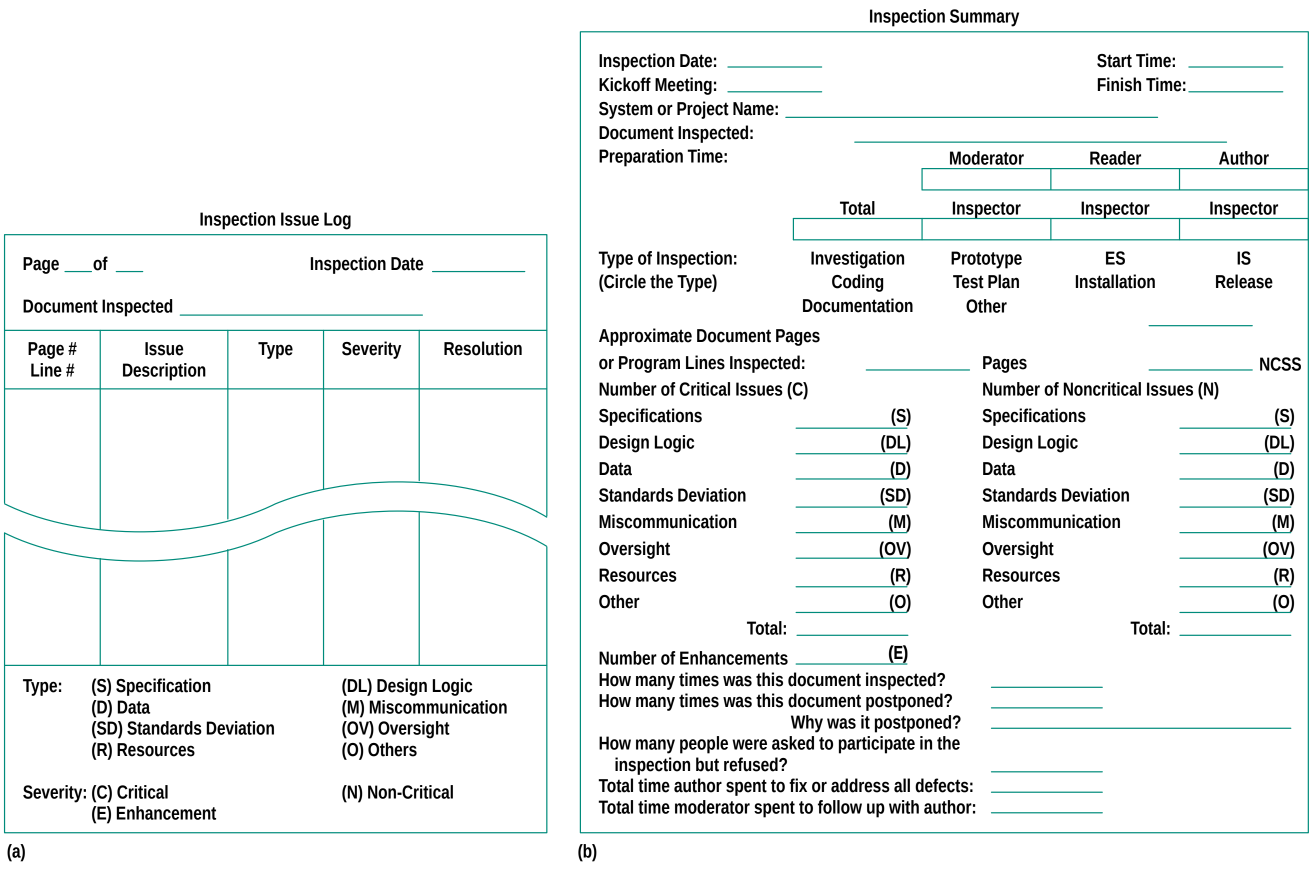
Inspection Metrics

Three forms were used to collect inspection metrics: the inspection issue log, the inspection summary form, and the inspection data summary and analysis form. Fig. 3 shows an inspection log and an inspection summary form.

Modules	Test Plan and Test Script Inspected	Percent of Programs Inspected	Percent of Job Streams (JCL) Inspected	Relative Complexity
1.0	Yes	67%	100%	Medium
2.0*	No	100%	100%	Low
3.0	Yes	60%	33.3%	Medium
4.0	Yes	100%	100%	High

* This module consisted of only one JCL and one program.

Fig. 2. Inspection coverage for the major modules in the SIT system based on the criteria used in the master plan.



Inspection Issue Log

Page ___of ___ Inspection Date __________

Document Inspected ______________________

Page # Line #	Issue Description	Type	Severity	Resolution

Type: (S) Specification (DL) Design Logic
(D) Data (M) Miscommunication
(SD) Standards Deviation (OV) Oversight
(R) Resources (O) Others

Severity: (C) Critical (N) Non-Critical
(E) Enhancement

(a)

Inspection Summary

Inspection Date: __________ Start Time: __________
Kickoff Meeting: __________ Finish Time: __________
System or Project Name: ______________________
Document Inspected: ______________________
Preparation Time:

		Moderator	Reader	Author
	Total	Inspector	Inspector	Inspector

Type of Inspection: (Circle the Type)	Investigation Coding Documentation	Prototype Test Plan Other	ES Installation	IS Release

Approximate Document Pages or Program Lines Inspected: __________ Pages __________ NCSS

Number of Critical Issues (C)		Number of Noncritical Issues (N)	
Specifications	(S)	Specifications	(S)
Design Logic	(DL)	Design Logic	(DL)
Data	(D)	Data	(D)
Standards Deviation	(SD)	Standards Deviation	(SD)
Miscommunication	(M)	Miscommunication	(M)
Oversight	(OV)	Oversight	(OV)
Resources	(R)	Resources	(R)
Other	(O)	Other	(O)
Total:		Total:	

Number of Enhancements __________(E)
How many times was this document inspected? __________
How many times was this document postponed? __________
Why was it postponed? __________
How many people were asked to participate in the inspection but refused? __________
Total time author spent to fix or address all defects: __________
Total time moderator spent to follow up with author: __________

(b)

Fig. 3. (a) Inspection issue log. (b) Inspection summary form.

The inspection issue log is used for logging the issues observed by the inspectors during the Issue and Question Logging session. The inspection summary describes the document inspected, the inspector's preparation time, the type of inspection , the number of pages and lines inspected, the number and types of defects identified, and the total time used to fix or address all the defects. The inspection data summary and analysis form is a spreadsheet that was used to collect the data entries required to calculate inspection efficiency, inspection effectiveness, total time saved, and the return-on-investment value (described later). Table III lists the data collected in the data summary and analysis form for each item inspected.

We selected four key inspection metrics to measure our inspection effort: number of critical defects found and fixed, number of noncritical defects found and fixed, total time used by inspections, and total time saved by inspections.

Testing Process

Our testing process included test planning, unit testing, module testing, and system testing. Test planning involved creating a master test plan and doing a risk assessment to determine where to focus our testing time.

Master Test Plan. A master test plan was created during the design phase when the test strategy for the project was outlined (Fig. 4). The master test plan included the test plan design and the type of tests to be performed. For design purposes, the system was divided into logical modules with each module performing a specific function (see Fig. 1). The test design was also oriented around this division.

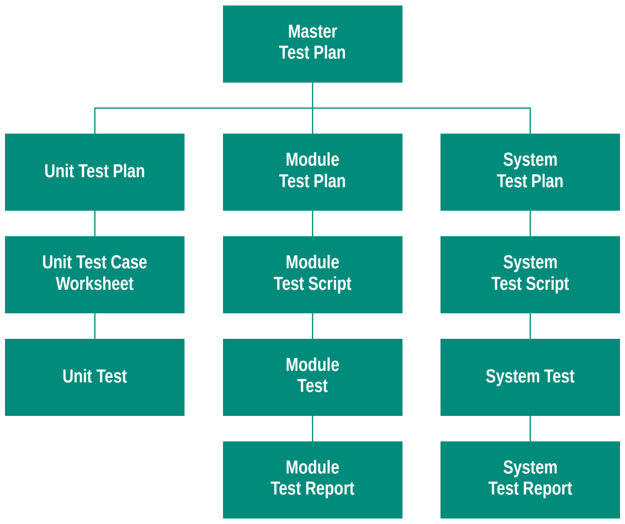


Fig. 4. Master test plan organization.

Table III
Inspection Summary and Analysis Data

Metric	Units or Source of Data
Inspection time	Preparation and meeting time in hours
Defects	Number of critical and noncritical defects
Documentation type	Code, requirements and design specifications, manuals, test plans, job streams (JCL), and other documents
Size of document	KNCSS for code and number of pages for other documents
Amount inspected	KNCSS or number of pages inspected
Preparation rate	= (number of pages) × (number of people)/preparation time
Logging rate	= (critical + noncritical defects)/ (hours/number of people)
Moderator follow-up time	Hours
Time to fix a defect	Hours
Total time used	= inspection time + time to fix + follow-up time
Time saved on critical defects	*
Time saved on non-critical defects	*
Total time saved	*
Return on investment	*

* Defined later in this article.

The primary and secondary features to be tested were also included in the master test plan. The primary features were tested against the product specifications and the accuracy of the data output. Secondarily, testing was performed to ensure optimum speed and efficiency, ease of use (user interface), and system supportability.

Risk Assessment. A risk analysis was performed to assess the relative risk associated with each module and its components. This risk analysis was used to help drive the schedule and lower-level unit and integrated tests. Two factors were used in assessing risk: operational importance to overall system functionality and technical difficulty and complexity. Each module was divided into submodules and rated against each risk factor. For example, the proper execution of the logic in submodule 4.2 was critical to the success of the system as a whole, while submodule 1.4 merely supplied additional reference data to the database. Accordingly, module 4.2 received a higher operational importance rating. Similarly, submodule 4.2 also received a higher complexity rating because of the complexity of the coding task it entailed. Each rating was based on a scale of one to five with one being the lowest rating and five being the highest rating. Ratings for each risk factor were then combined to get an overall risk rating for each submodule (Fig. 5).

Module	Submodule	Operational Importance	Technical Difficulty	Overall Risk Rating
1.0	1.1	5	1	6
	1.2	5	3	8
	1.3	5	3	8
	1.4	1	1	2
	1.5	4	3	7
	1.6	3	2	5
2.0	2.1	5	2	7
	2.2	5	3	8
3.0	3.1	5	4	9
	3.2	5	2	7
	3.3	3	2	5
	3.4	5	2	7
	3.5	4	5	9
	3.6	1	1	2
4.0	4.1	5	3	8
	4.2	5	5	10
	4.3	5	4	9
	4.4	5	3	8
	4.5	3	2	5
	4.6	2	4	6
	4.7	1	3	4

Fig. 5. Risk ratings by submodule.

Unit Testing. Unit testing, as in most software projects, was performed for all programs and job streams. For the purpose of this project, each individual program and job stream was considered a unit. Since programs were often embedded in job streams, program unit tests were often synchronized with job stream unit tests to conserve time and effort.

Because of the small size of the project team, almost all tests were performed by the program or job stream author. To minimize the impact of this shortcoming, a simple testing review process was established. A series of standard forms were created to document each test and facilitate review by the designer, users, and the project lead at different points in the unit testing process (see Fig. 6).

Module Testing. An integrated test of all programs and job streams within each module was conducted to test the overall functionality of each of the four system modules. Since each program or job stream had already been tested during unit testing, the primary focus of module testing was on verifying that the units all functioned together properly and that the desired end result of the module's processing was achieved.

A brief integrated test plan document was created for each module. This test plan listed the features to be tested and outlined the approach to be used. In addition, completion criteria, test deliverables and required resources were documented (Fig. 7).

Detailed test scripts were used to facilitate each module test and make it easy to duplicate or rerun a particular test. Each

Unit Test Case Worksheet

Module Unit: 4.2

Program/Screen/Job Name: SIT4023S

VALID Input Conditions	INVALID Input Conditions
Loc_hdr.id_code<>9	No transaction header
Quantity qualified = 14, 17, 76	ANSI code < > 867, 846
	Tx_count = spaces

Unit Test Script

Module Unit: 4.3

Program/Screen/Job Name: SIT 4031S

Script #:

Procedure
(1) Set file equations
(2) Run program, parm = c
(3) Verify file layout matches
Resources needed —
Programs: GETPROD
Screens: —
Database: SITDB, PRIME
SITDB Tables: Outlets, Channel_products
Others

Unit Test Case

Module Unit: 4.2 **Date Inspected:** 5/4/92

Program/Screen/Job Name: SIT 4020J

Script #:

Pass: [✓ Yes] / No

Case Description	
Verify that locations with missing elements are reported.	
Input Conditions	
Outlets table "business_name" blank	OID = 0671100920
End-user table "company" blank	OID = 067110011S
Output Conditions	
OID = 0671100920 appears on report	
OID = 067110011S appears on report	
Special Requirements	
Use test file TST40207.TEST.SIT	

Fig. 6. Unit test forms.

Feature to Be Tested
- **Module processes run correctly when run in production order**
- **Module processes run correctly with actual production data**
- **All programs, JCLs, screens pass data to next process on timely basis**

Approach
- **Set up production test environment**
- **Stream JCLs, run programs, and execute entry screens in production order**
- **Use production data**
- **Verify subsets of key table data after each process**

Completion Criteria
- **All programs, JCLs, screens run without error or abort**
- **End result of process is correct data loaded into Product_mix, Product_exhibit, Channel_products, Customer_Products, and Customer_Exhibits Tables**

Test Deliverables
- **Test script**
- **Test summary report**

Resources
- **Staff – Lou, Shripad, Jill**
- **Environment**

Jupiter	**– SIT account**
Blitz	**– Patsy DB (PATSY##.PATDTA.MAS):Mode 5**
	– Prime DB (PRIME3##.PR3DTA.MAS):Mode 6

Fig. 7. A portion of an integrated test plan.

test script document included a listing of the resources needed for the test, such as supporting databases, SIT database tables, files, file equations, programs, and a step-by-step procedure for executing the test. Where appropriate, specific data to be entered into the system was included in the procedure. Also included were the expected results of each test step and the overall test.

Once the integrated module tests had been completed successfully (often this took several iterations), a report detailing the test results was created. The integrated test report documented the number of times the test was run, verified that the completion criteria were met, listed the number of critical and noncritical defects detected, and verified that each defect had been fixed.

System Testing. System testing was conducted in tandem with a pilot test at an HP dealer. Initial values were entered for the general-purpose database tables and for the dealer involved in the pilot test. Production schedules were used to control the job streaming that executed the system.

Testing Metrics. Two kinds of metrics were selected to measure our testing effort: total number of critical defects and total testing time for each test phase. Table IV summarizes our test metrics for unit and module testing. One critical defect was found during system testing and the total system test time was 19 hours.

Table IV
Testing Metrics

Module	Size*	Unit Test			Module Test		
		T_c	N_c	$(ATT)_c$	T_c	N_c	$(ATT)_c$
1.0	4489	73	4	18.25	41	7	5.86
2.0	639	26	0	0	**	**	**
3.0	3516	31.5	3	10.5	46	2	5.11
4.0	3738	35	21	1.67	36	13	2.77

* Noncomment source statements (NCSS)

** Not applicable

T_c = Total testing time for critical defects in hours

N_c = Total number of critical defects

$(ATT)_c$ = Average testing time per critical defect = T_c/N_c
$(ATT)_c$ is defined as zero when N_c is zero

Return-on-Investment Model

It is now generally accepted that inspections can help software projects find defects early in the development cycle. Similarly, the main purpose of unit and module testing is to detect defects before system or pilot testing. Questions that often come up regarding these defect finding efforts include how much project time they will consume, how effective they are, and how we can measure their value. Most of these issues have been addressed in different ways in the literature.[4,5]

In this section we present a return-on-investment (ROI) model we used to measure the value of inspections and early testing in terms of time saved (and early to market). The whole idea behind this kind of measurement is that it should take longer to find and fix a defect at system test than it does to find the same defect during inspection or unit testing. This means that for every defect found during an inspection or at an earlier stage of the testing phase, there should be a time savings realized at the system test phase.

First we define the Prerelease ROI as:

Prerelease ROI =
Total Time Saved / Total Time Used (1)

where:

Total Time Saved = Total Time Saved by Inspection + Total Time Saved by Unit and Module Testing

and

Total Time Used = Total Time Used by Inspection + Total Time Used by Unit and Module Testing.

We calculate the individual ROI for inspection and testing, respectively, as follows:

Inspection ROI = Total Time Saved by Inspection / Total Time Used by Inspection (2)

Testing ROI = Total Time Saved by Unit and Module Testing / Total Time Used by Unit and Module Testing. (3)

We wanted to measure not only how much time was being spent on inspection and testing but also how much time was being saved as a result of the defects found during inspections and unit and module testing.

The time used during an inspection includes the sum of the total inspection time spent by each team member, the time spent by the author on fixing the defects, and the time spent by the moderator following up on defect resolution with the author.

For inspections, we defined the total time saved and the total time used as:

Total Time Saved by Inspection = Time Saved on Critical Defects + Time Saved on Noncritical Defects

Total Time Used by Inspection = Inspection Time + Time to Fix and Follow up for Defect Resolution

where the critical defects are defects that affect functionality and performance and noncritical defects are all other defects.

The time spent finding and fixing a critical defect at system test is called BBT (black box testing time). Therefore, for every critical defect found before system test, the total time saved can be calculated as follows:

Time Saved on Critical Defects = BBT × Number of Critical Defects − Total Time Used.

The model we used to measure noncritical defects is based on the assumption that noncritical defects could be found by inspection but would not be detected by testing. The noncritical defects will become supportability issues after manufacturing release. We defined a new variable called MTTR* (mean total time to rework) to measure the time spent on noncritical defects.

MTTR = Time to Find Defect + Time to Fix Defect + Time to Release to Production.

Thus,

Time Saved on Noncritical Defects = MTTR × Number of Noncritical Defects.

For testing metrics we wanted to measure not only how much time was being spent on unit and module testing, but also how much time was being saved as a result of the defects found during these tests. Thus, we defined the total time saved and total time used for testing as:

Total Time Saved = Time Saved on Critical Defects

Total Time Used = Time to Design and Build a Test + Time to Execute + Time to Find and Fix a Defect.

* We used an average time of 6 hours for MTTR in our calculations.

The defect data and time data for our sales and inventory tracking project are summarized in Tables V and VI.

Table V
Defect Summary for the SIT Project

	During Inspection	During Testing	Total Prerelease Defects
Number of Critical Defects Found and Fixed	12	51	63
Number of Noncritical Defects Found and Fixed	78	0	78

With the code size equal to 13.6 KNCSS, prerelease defect density = 141/13.6 = 10.4 defects/KNCSS.

Using the model described above, we can calculate the ROI values shown in Table VI.

Total Time Saved by Inspection = (20 hours × 12) + (6 hours × 78) − 90 hours = 708 hours**

Total Time Saved by Testing = 20 hours × 51 − 310 = 710 hours*

From equations 1, 2, and 3:

ROI for Inspections = 708 / 90 = 787%

ROI for Testing = 710 / 310 = 229%

Prerelease Total ROI = 1418 / 400 = 355%.

Table VI
Time Data and Return on Investment Results

	Total Time Used (hours)	Total Time Saved (hours)	Return on Investment (%)
Inspection	90	708	787
Testing	310	710	229
Prerelease Total	400	1418	355

Results

Fig. 8 shows that with the exception of module SIT4.0 the average testing time per critical defect decreased from unit test to module test for the system's major modules. The reason that it took 19 hours per critical defect at system test is mainly the time it took to find and fix one defect that was overlooked during inspection. Had the project team not overlooked one particular issue related to product structure that resulted in this defect, the average testing time per critical defect at system test would have been significantly lower.

Module SIT4.0 went through the most thorough inspection including a design inspection since it is the most complex of the four modules. We believe our efforts paid off because it took less time at unit test and module test in terms of average testing time per critical defect for module SIT4.0 than for the other three modules.

* The time to find and fix a critical defect during system test at HP ranges from 4 to 20 hours. We used 20 hours in our ROI calculations.

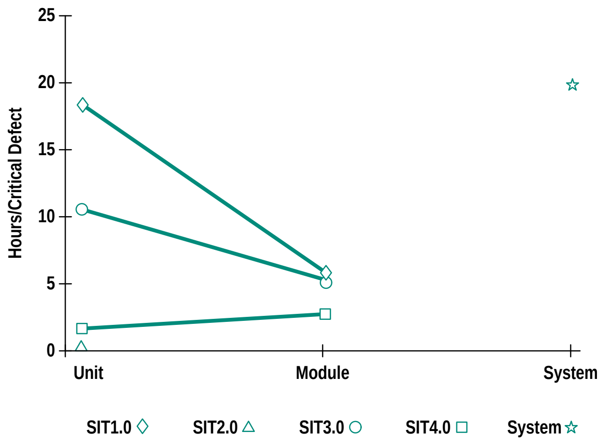


Fig. 8. Testing time by test phase and module.

Fig. 9 is a plot of the ROI column in Table VI. It shows that inspections have resulted in more than three times the ROI of testing. This reinforces the notion that a great deal of money and time can be saved by finding defects early in the software development cycle.

Lessons Learned

The inspection and testing processes we used for the SIT project are not very different from other software projects in HP. However, we did put more emphasis on early defect detection and collected a lot of metrics. The following are some of the lessons we learned from our efforts during this project.

Project Management. Some of the lessons we learned about project management include:

- Setting aside time for inspections and thorough testing does pay off in the long run. Management approval may be difficult to get, especially when under intense time pressure. One should get commitment to delivering a quality product, then present inspections and testing as part of delivering on this commitment.
- Keep high-level test plans short and simple while still providing enough direction for the lower-level plans. By keeping these plans short and simple, time will be saved and the project team can still get adequate direction.
- Adequate follow-up to inspection and testing activities is important to make sure all issues are resolved. The inspection log, testing error logs, and integration test reports

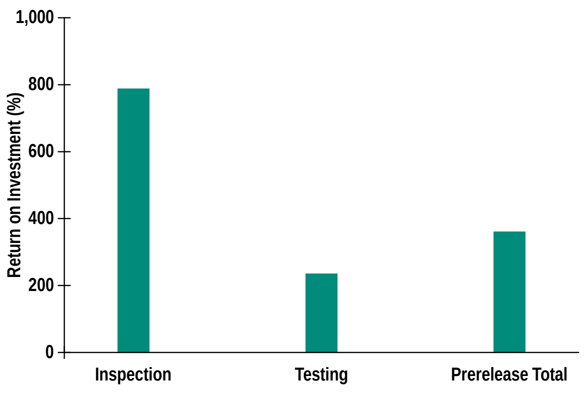


Fig. 9. Relative impact of inspections and testing.

helped the project lead keep up on the status of each issue and ensure that each issue was resolved.
- Establish coding standards at the beginning so that minimal time is spent during code inspections questioning points of style. The focus of code inspections should be code functionality.

Inspection. Since the inspection process is the most important tool for defect-free software, many adjustments were made here.
- Attitude is the key to effective inspections. No one writes error-free code, but many people think they do. Authors must realize that they make mistakes and take the attitude that they want inspectors to find these errors. The inspectors, on the other hand, can destroy the whole process by being too critical. The inspectors must keep the author's ego intact by remaining constructive. Perhaps the best way to keep people's attitudes in line is to make sure they know that they may be an inspector now, but at a later date, their role and the author's will be reversed. For this reason, implementing an inspection process for most or all of a project's code is likely to be much more effective than random inspection of a few programs.
- No managers should be involved in the inspection of code. Having a manager present tends to put the author on the defensive. Also, depending on the person, the inspector either goes on the offensive or withdraws from the process entirely.
- In addition to finding defects (or "issues"), which helps to save testing and rework time, the inspection process has other, more intangible, benefits:
  - Increased teamwork. Inspections provide an excellent forum for the team to see each other's strengths and weaknesses and gain a new respect for each other's unique abilities. By adding the question and answer session to the inspection process, we provided a forum for the team to discuss issues and creatively solve them together.
  - Support team education. Including members of the team that would eventually support the SIT system allowed these people to become familiar with the system and confident that it would be supportable.

Testing. The lessons learned from unit and module testing include the need for expanded participation in testing and the value of test scripts.
- Unit testing. On small project teams it is difficult to coordinate testing so that someone other than the author tests each unit. Establishing a process that includes the designer, other programmers, and users helps tremendously towards ensuring full test case coverage.
- Module testing. Integration test scripts are invaluable. The effort expended to create the scripts for the SIT project was significant, especially the first one. However, the reward, in terms of time saved and rework, more than justified the effort. Furthermore, these scripts have been very useful to the support team for performing regression testing when the programs or job streams are modified.

Success Factors. The SIT product was released to production in early March 1992. Since that time the product has been relatively defect-free. In reviewing what has been done, we observed some key factors that contributed to our success. These success factors can be summarized as following:
- Strong management support. We had very strong management support for the inspection and testing process and the time commitment involved. This was the most important and critical success factor for the implementation of inspections and metrics collection.
- Team acceptance. The SIT project team accepted the quality concept. We agreed on our quality goals and understood how the inspection and testing processes would help us to achieve those goals.
- Focus. The SIT project was selected as the pilot project to implement the inspection process. Our initial focus was on code inspection. After the project team felt comfortable with doing inspections, other documents such as test scripts and test plans were also inspected.

Acknowledgments

We would like to thank the following people for their help and support in developing this paper: Jennifer Hansen, Tadd Koziel, Bruce Morris, Patti Hutchison, John Robertson, Debra Vent, and Kelley Wood.

References

1. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM System Journal,* Vol.15, no. 3, 1976, pp. 182-211.
2. M.E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering,* Vol. SE-12, no. 7, July 1986, pp. 744-751.
3. T. Gilb, *Principles of Software Engineering Management,* Addison-Wesley Publishing Co., 1988, Chapter 12.
4. F.W. Blakely and M.E. Boles, "A Case Study of Code Inspections," *Hewlett-Packard Journal,* Vol. 42, no. 4, October 1991, pp. 58-63.
5. W.T. Ward, "Calculating the Real Cost of Software Defects," *Hewlett-Packard Journal,* Vol. 42, no.4, October 1991, pp. 55-58.