Databorough

A Visual Guide to Cost Effective Testing on the iSeries

Steve Kilner

A Visual Guide to Cost Effective Testing on the iSeries

Managers and developers working with legacy RPG and COBOL applications face serious testing challenges. Few, if any of these applications have a complete set of specifications of how the applications should perform, and even fewer have a complete library of test cases.

The majority of the work being done on iSeries applications involves enhancements and corrections. This type of work requires regression testing the modified code to ensure that new defects have not been introduced to production systems. Without a complete library of test cases, testing becomes increasingly difficult, expensive and risky, leading to the following unfortunate truth for most iSeries development projects that lack good testing capabilities:

Results from over 12,000 software projects show that detecting and removing defects is the single most costly task in software development.

Regression testing is either the biggest task
in a project, or the biggest risk.

This paper describes the variety of tools that can be used to improve testing productivity and results for both regression testing and new functionality testing. It is divided into the following sections:

The Business Case For Investing In Testing

From 50 years of statistics on over 12,000 software projects it has been clearly shown that the single most costly aspect of software development is detecting and correcting defects, comprising roughly 35 percent of

Defects found in production – testing failures – are 5 to 10 times more costly to fix.

total software development and maintenance costs.[1]
Of key interest to IT managers, is the statistic that *defects found in production are 5-10 times more costly to fix* than defects found in the coding stage. It doesn't take too many of those defects to become a signficant factor in IT budgets.

While large statistical studies of RPG defect rates are hard to come by, using defect rates per function point, and extrapolation of lines of code per function point for RPG, it is possible to estimate that there are roughly eight defects per hundred lines of code changed or written.

Across industries, the average rate of change per year in application code bases is roughly seven percent. Using these figures provides the basis for estimating the number of defects that IT must deal with.

Of general interest are the figures for the point in the software lifecycle where defects originate:

Defect sources	**Percent**
Requirements errors	20
Design	25
Coding	35
Documentation	12
Bad fixes	8

Also of interest is the pattern of defect detection and removal. Most IT managers understand that testing never detects all defects, but what is the actual pattern of detection?

Defect detection proceeds in a manner similar to that of the half-life decay of radioactive elements. For example, on average, unit testing will detect 25% of the resident defects. If unit testing were performed a second time it would detect 25% of the remaining defects.

Key Fact

Testing is never perfect and produces diminishing returns, *therefore it is crucial to optimize testing efficiency.*

The first step towards testing efficiency

By bringing practical tools and advanced techniques to how you test, the costs and risks associated with testing can be reduced dramatically, measurably improving IT throughput and saving scarce IT dollars.

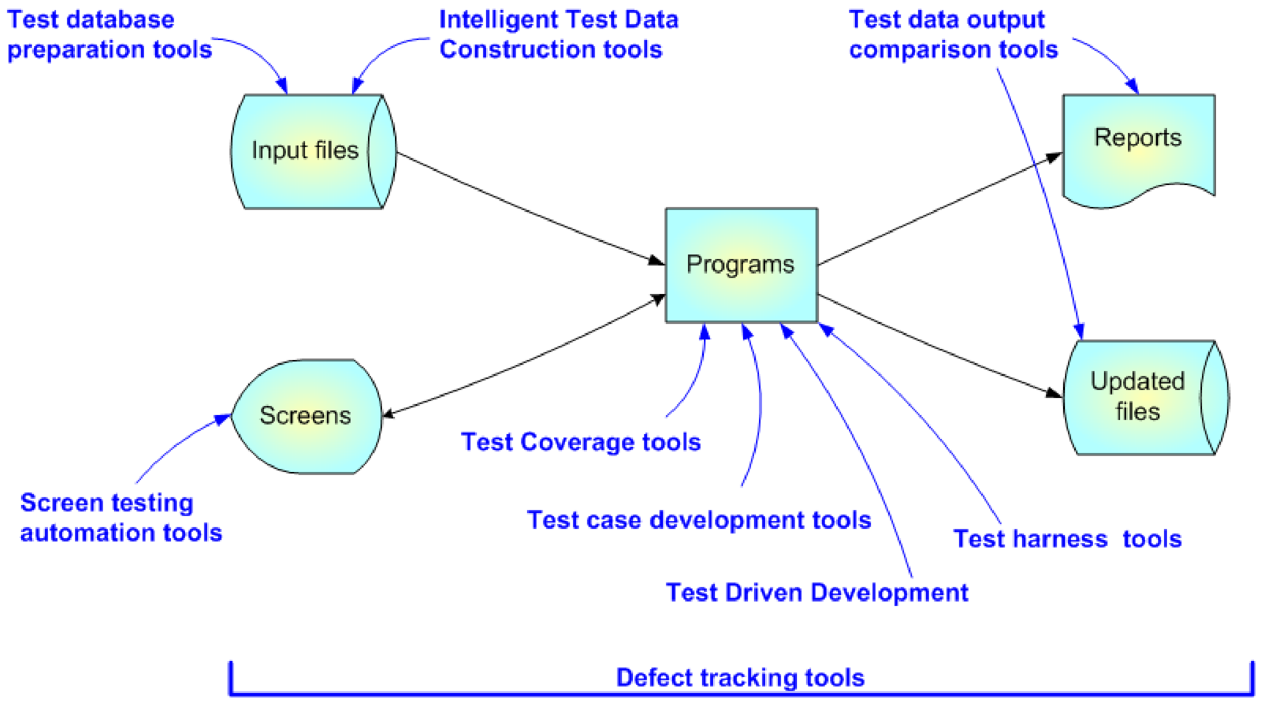
Extreme Testing of Legacy Applications As A Strategy

One of the fundamental strategies of testing legacy applications is to perform high volume, or "extreme," testing, enabled by the use of powerful tools and testing techniques. Such testing is the best substitute available for regression testing in the absence of complete libraries of test cases or application specifications.

In particular, wherever possible, run *high volumes of data through parallel tests*. This requires the use of a number of tools for tasks such as preparing databases, comparing results, creating test harnesses and so on, as described below.

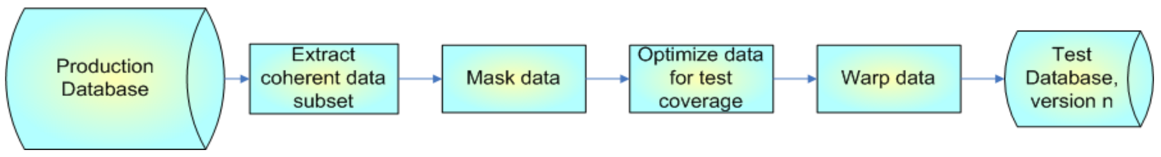
Overview Diagram of Testing Tool Applications

The diagram below provides an overview of where tools can be applied to your testing processes.


Test database preparation tools
Intelligent Test Data Construction tools
Test data output comparison tools
Input files
Reports
Programs
Updated files
Screens
Test Coverage tools
Screen testing automation tools
Test case development tools
Test harness tools
Test Driven Development
Defect tracking tools

Test Database Preparation Tools

Purpose: create a test database that is ideal for regression testing or new functionality testing.



Typical functions:

- Extract a subset of records from production tables such that the subset is relationally coherent, i.e., all foreign keys have relationally valid parent records.

  For example, if you are testing changes related to cancelled orders you might first build a subsetted order header file of cancelled orders. You would then include the related order detail records. And then you would have to include related customer records, item records, code field records for all codes on all those tables, and so on. This can become quite involved and is time-consuming and error prone if done manually. A full-featured tool with complete knowledge of the data model and foreign key relationships can automate the entire process.

- Mask data, to protect consumer privacy

  It is often necessary to mask data such as consumer names, government ID numbers, phone numbers, addresses and so on. For testing purposes it is only feasible to do this if appearances are maintained so that the information displayed looks like the type of information it is.

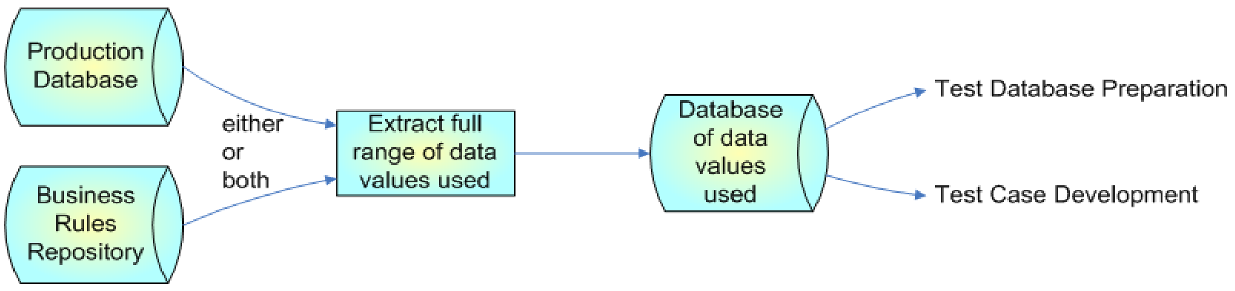
  It is also sometimes necessary to mask sensitive internal data, particularly if it is sent offsite to outsourced staff.

- Optimize test data for maximum test coverage, i.e., ensure that the test data set contains all possible values – see more about this in the next section, *Intelligent Test Data Construction Tools*.

- Warp data, most commonly to move dates forward and backwards for more complete test coverage of time-sensitive functionality.

Intelligent Test Data Construction Tools

Purpose: to build an optimal set of test data to ensure maximum coverage of all test cases and logic paths.

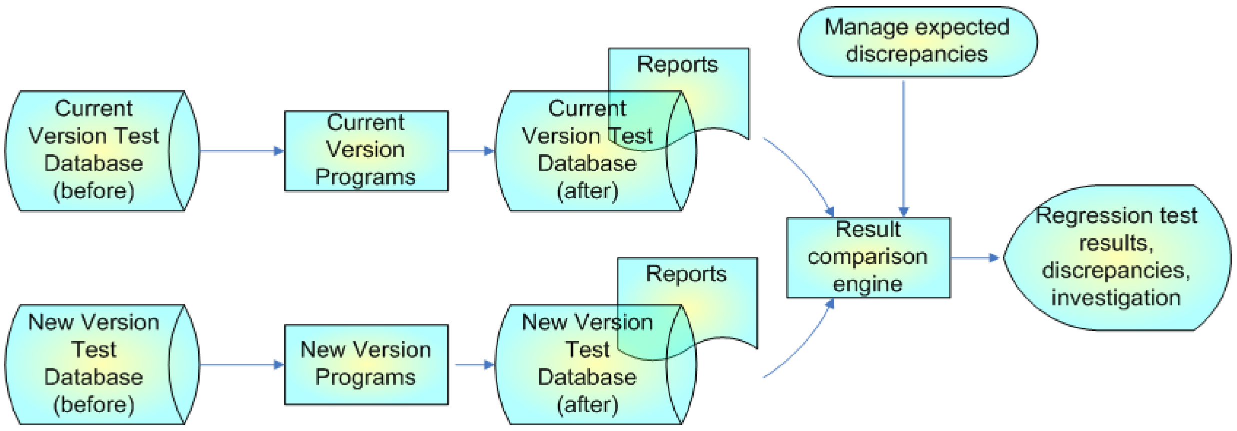

Production Database
Business Rules Repository
either or both
Extract full range of data values used
Database of data values used
Test Database Preparation
Test Case Development

This kind of tool can use the results of one or both of two kinds of analysis: 1) analysis of existing values in the database for fields in tables, or 2) analysis of the fields used in business rules. In either case, the purpose is to diminish gaps in test case coverage and enumerate data values to be included in test cases.

The data produced by this type of tool is input to either or both *of Test Database Preparation Tools* or *Test Case Development Tools*.

Test Data Output Comparison Tools

Purpose: For regression testing, to compare the output of new, modified code against the output of current production code.


Current Version Test Database (before)
Current Version Programs
Current Version Test Database (after)
Reports
Manage expected discrepancies
Result comparison engine
Regression test results, discrepancies, investigation
New Version Test Database (before)
New Version Programs
New Version Test Database (after)
Reports

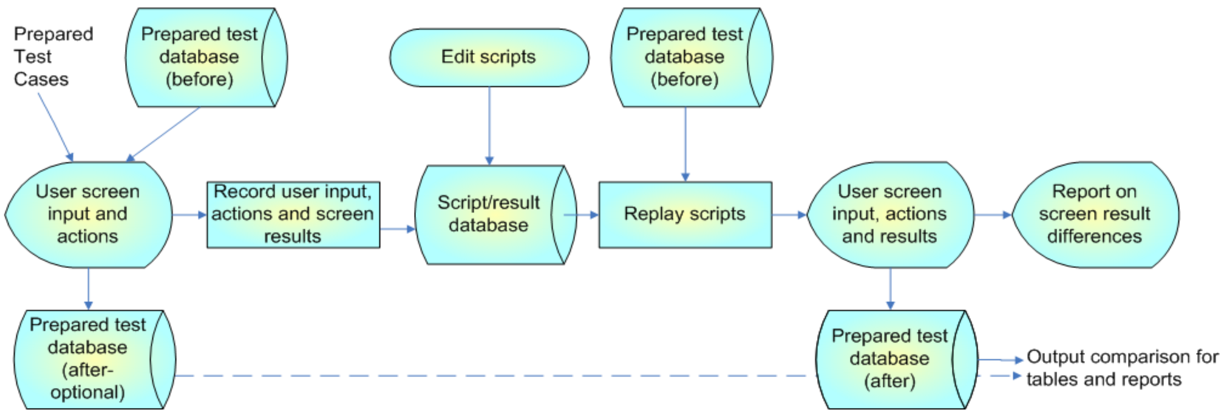
These tools compare the output of current versions of programs against the output of modified versions of the same programs.

Typically this is done by having two versions of tables and output queues in different libraries and then comparing them on a record by record, field by field basis. Discrepancies are reported. Full featured tools allow quick navigation and drill-down into those discrepancies to research the thousands of possible fields.

There is also typically a facility for filtering out expected, or unimportant, discrepancies, such as timestamps for example, to help focus on meaningful discrepancies.

Screen Testing Automation Tools

Purpose: for regression testing, to re-execute large numbers of screen interactions on demand in parallel, or for functional testing to replay test cases as needed.


Prepared Test Cases
Prepared test database (before)
Edit scripts
Prepared test database (before)
User screen input and actions
Record user input, actions and screen results
Script/result database
Replay scripts
User screen input, actions and results
Report on screen result differences
Prepared test database (after-optional)
Prepared test database (after)
Output comparison for tables and reports

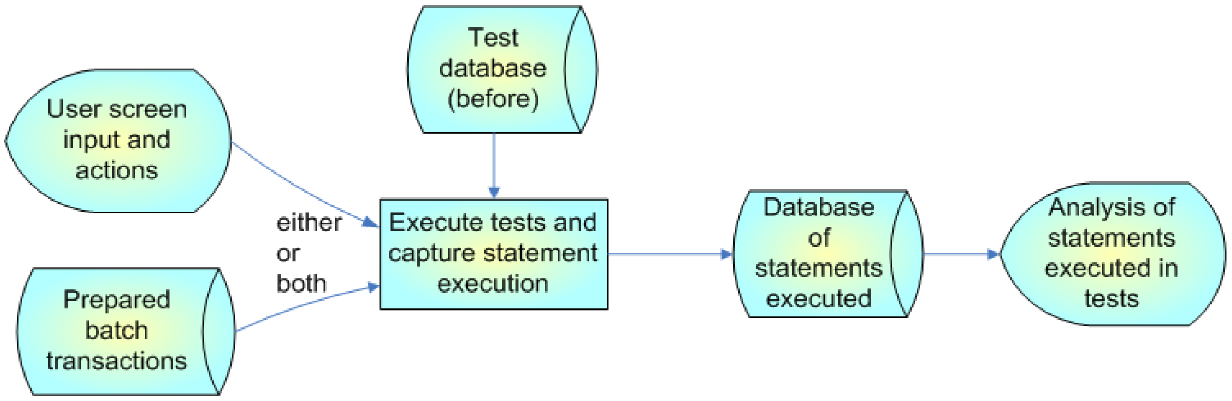
Typical practice is to 1) capture input keystrokes and mouse movements, and 2) capture and compare results of events, sometimes limited to screen results.

Usually also requires a means of repeatedly creating a test database in a particular state, and also writing well-defined test cases ready for the interactions. With interactive testing there are usually at least three possible outputs needing comparison for regression testing: screens, database fields and reports.

Automating screen testing is time-consuming and complex. It requires a foundation of test database preparation tools, as described above. It is typically better to begin by improving other aspects of test automation first.

Test Coverage Tools

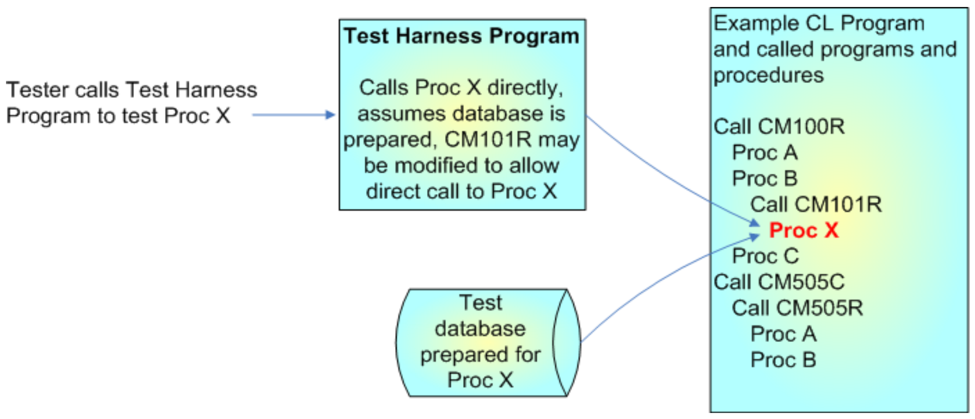
Purpose: to track which lines of code have been executed in tests, in order to assure more complete test coverage.


User screen input and actions
Prepared batch transactions
either or both
Test database (before)
Execute tests and capture statement execution
Database of statements executed
Analysis of statements executed in tests

Typically these track which lines of code have been executed and how many times, not the variations of paths through the code, which is far more complex. Despite the unlikely prospect of detecting and tracking all possible paths, merely observing that certain lines of code were never executed can help reveal serious gaps in test cases, or even reveal defects themselves.

Test Harness Tools

Purpose: enable the calling of programs, procedures or subroutines, not normally at the top of calling hierarchies, or that have special requirements for calling.

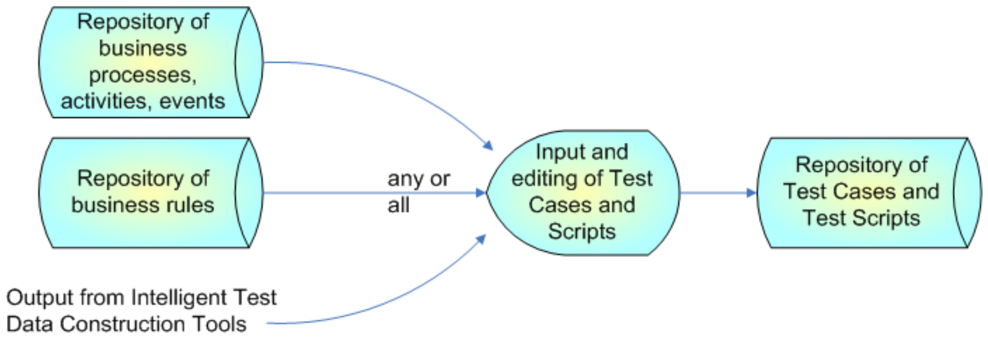

Tester calls Test Harness Program to test Proc X
Test Harness Program
Calls Proc X directly, assumes database is prepared, CM101R may be modified to allow direct call to Proc X
Test database prepared for Proc X
Example CL Program and called programs and procedures
Call CM100R
Proc A
Proc B
Call CM101R
Proc X
Proc C
Call CM505C
Call CM505R
Proc A
Proc B

These tools enable more efficient and focused testing, particularly of modifications. In the RPG world these are usually customizations created for project purposes.

Using a code analysis tool greatly facilitates formulation of test harnesses by revealing information necessary for interfacing between modules.

Test Case Development Tools

Purpose: to allow entry of test cases and test scripts, working in concert with facilities that analyze legacy code and assist testers with developing a more complete set of test cases.


Repository of business processes, activities, events
Repository of business rules
any or all
Output from Intelligent Test Data Construction Tools
Input and editing of Test Cases and Scripts
Repository of Test Cases and Test Scripts

The functionality can be broken into three parts, which may be separate tools:

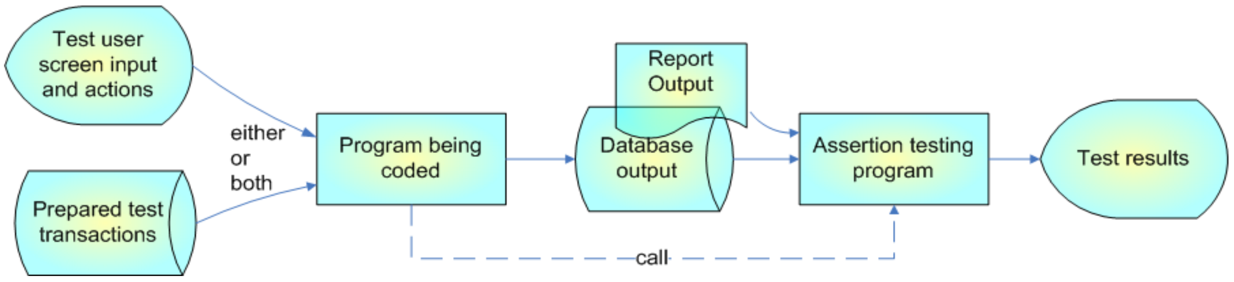
- Analysis of the existing system – this usually applies to modified legacy systems and regression testing and is facilitated by the availability of a repository of business processes, activities, business rules, event, etc. X-Analysis is an example of such a tool.
- Output from Intelligent Test Data Construction Tools – described in an earlier section, this tool provides a comprehensive view of the range of test data values necessary to execute a full test of the application
- Input and editing of test cases and test scripts – test cases and scripts are defined and entered through a GUI based on information from a repository, as described above, the requirement specifications of the project being tested, and the range of values necessary for complete testing.

Tools that comprise an analytical repository typically gather and display any or all of three types of information:

- Events that occur in the program that testers should be sure to evoke
- Conditional logic in the program that lead testers to the test data they need to create in order to test the conditions; these are often business rules
- Data output by the program that lead testers to the data they need check to prove the test results were correct

Test Driven Development

Purpose: build testing directly into the development process


Test user screen input and actions
Prepared test transactions
either or both
Program being coded
Report Output
Database output
Assertion testing program
Test results
call

Using this methodology, developers begin by writing code that tests whether the new code they are about to write is working correctly. For example, if a new function is being created that calculates the total tax amount on order detail records and updates the order header record with that total, then the test code checks to see whether the totals have been accumulated and updated correctly. This is often termed an *assertion*.

The concept is that developers first define all the tests and keep coding and running the tests until the new code produces correct test results.

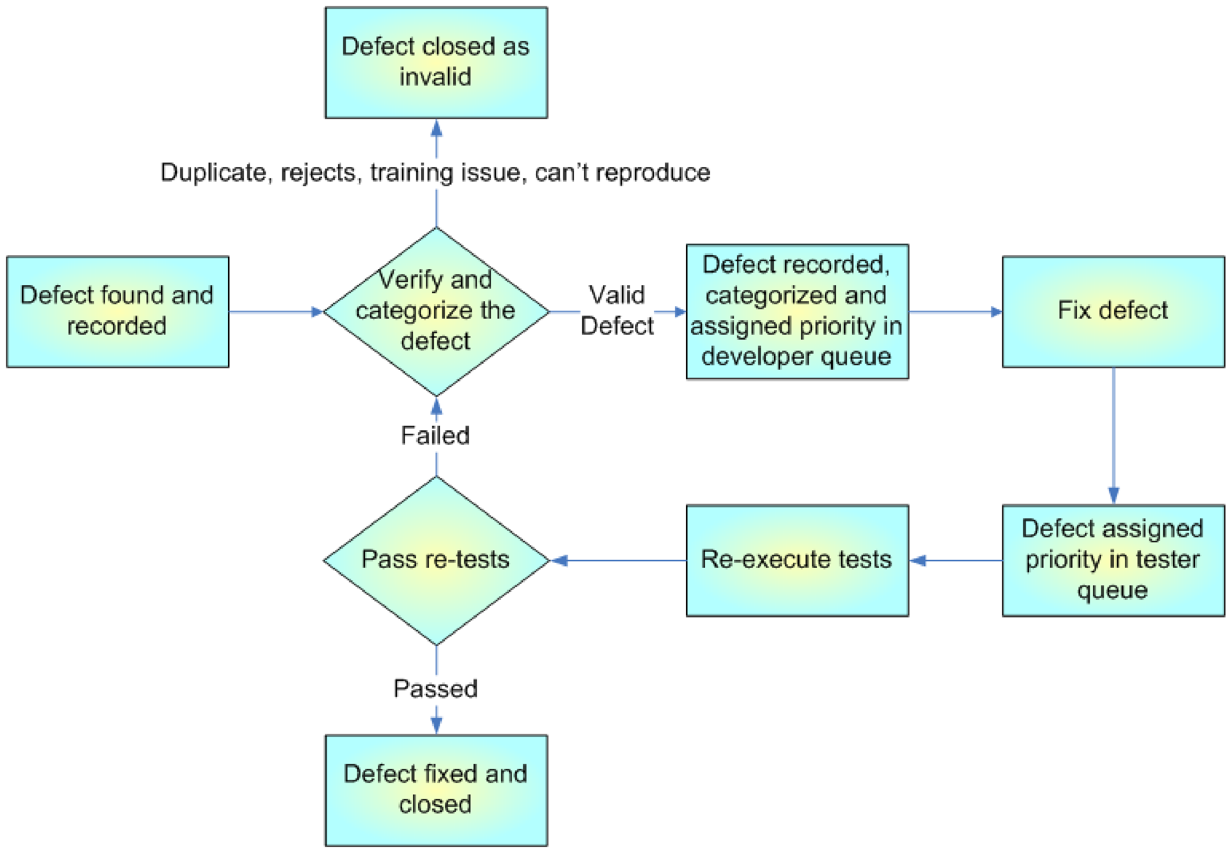
There are different implementation approaches; the approach shown above consists of a separate module containing all the test cases. This is called when the program being developed completes execution. Another approach is to call individual tests from within the program being coded, with the calls being disabled upon deployment through use of an environmental switch.

In some languages developers build the test cases directly into their code. The RPG Unit open source library provides tutorials and examples.

Test driven development is used most commonly in greenfield agile projects, but its suitability for legacy applications is unproven.

Defect Tracking Tools

Purpose: to track defects that make it out of development into QA and/or production.


Defect closed as invalid
Duplicate, rejects, training issue, can't reproduce
Defect found and recorded
Verify and categorize the defect
Valid Defect
Defect recorded, categorized and assigned priority in developer queue
Fix defect
Failed
Pass re-tests
Re-execute tests
Defect assigned priority in tester queue
Passed
Defect fixed and closed

These tools typically allow defects to be recorded and assigned associated program names, root cause codes, responsible party names, screen names and other useful information.

The diagram shown depicts a typical cycle of recording defects and tracking the associated workflow. There are a number of both commercial and open source tools available for defect tracking.

By tracking defects it is possible to implement a continuous improvement process for developing specifications, coding, testing and deployment, leading to both the creation of fewer defects, and earlier detection, resulting in cost savings. improved quality and faster deployments.

Many studies have shown that defects found in production are more than five times as costly to fix as those found in development.

The Business Case Again

As noted in the comments the top of this paper, detecting defects (testing) and correcting them are the costliest part of the software lifecycle – even more so than developing specifications or coding.

Further, defects found early in the development process can be fixed at a mere 10-20% of the cost of fixing defects that reach production.

Improving testing, especially in the face of the challenge presenting by undocumented legacy systems, is one of the best values in process improvement available to iSeries IT managers.

Read more about Databorough's test suite for the iSeries, or arrange for a free trial.

[1] All statistics quoted from *Estimating Software Costs, Bringing Realism To Estimating*, 2007, Capers Jones