# Active Feedback Loops in Software Testing

**Robert Sabourin**

robsab@gmail.com

**Chris Blain**

cblain@gmail.com

## Abstract

Rob Sabourin and Chris Blain share many decades of experience implementing software engineering concepts in a wide variety of business, technical, organizational, and cultural contexts.

While testing we may miss opportunities to act. Can testing become a call to action? Can testers hold a leading role driving decisions and getting things done?

Testing can be focused on taking action.

Feedback loops can be established between testers and programmers, test leads, other testers, subject matter experts, scrum masters, product owners, customers, and users.

Feedback loops apply to traditional or “agile” methods and are particularly valuable in regulated projects. These loops can be more useful and motivating to teams versus traditional phase gates that teams often dread. If the feedback loop is seen as a way to improve and increase the quality of testing (and the entire project as a result), teams will be more engaged.

Feedback from the testers can guide programmers to improve code quality and adjust programming methods especially when a cluster of related bugs is identified. Feedback from programmers to testers can lead to revising the scope and depth of testing and better understanding technical risk. Feedback from testers to product managers can help the team understand what can really get “done-done” in a certain time period.

Feedback from product owners and customers can help testers understand business context factors which influence testing and test reporting.

Feedback from testers to product owners and customers can build confidence in product quality influencing go/no go deployment decisions.

Feedback to testers from users can offer a much-needed dose of realism to the scenarios and data exercised in a session. As well as the real usage of various features.

Testers can suggest improved tools and test design techniques.

All stakeholders can guide testers in how to make their findings more relevant.

## Biography

*Robert Sabourin has more than forty years of management experience, leading teams of software development professionals. A well-respected member of the software engineering community, Robert has managed, trained, mentored, and coached thousands of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. The author of “I am a Bug!” the popular software testing children’s book, Robert is an adjunct professor of Software Engineering at McGill University.  Robert is the principal consultant (&president/janitor) of AmiBug.Com, Inc.*

*Chris Blain has more than twenty years of experience working in software development on projects ranging from embedded systems to SaaS applications. He is a former board member of the Pacific Northwest Software Quality Conference and a semi-regular conference speaker. His main interests are testing, distributed systems, programming languages, and debugging.*

# 1 Introduction

This paper will describe several different approaches used by the authors' to implement active feedback related to testing activities in software engineering projects. Approaches have been successfully applied to a wide variety of process models including blends of agile, traditional and hybrid life cycles.

In this paper Robert Sabourin and Chris Blain will share several aspects of Active Feedback through examples taken from real software development projects.

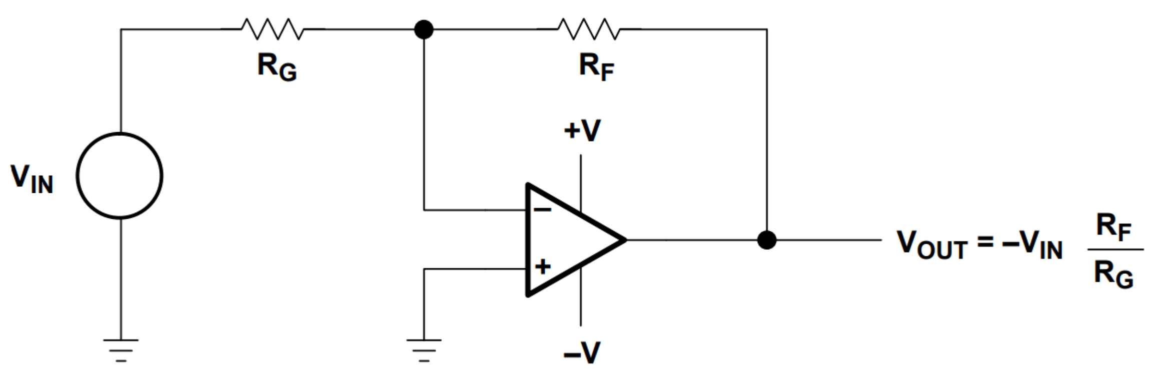
The lessons learned may be interesting to organizations and development professionals seeking to identify methods to improve their software engineering process and practice as a natural extension of the technical work being accomplished related to product requirements, designs, process and software quality assurance. In addition, ideas for improved training engagement with technical teams are contained as well.

# 2 Feedback Loops from Analogy to Reality

## 2.1 Feedback in Electronics

Feedback is a concept used in engineering to implement adaptive systems. The behavior of the system is designed to adapt based on system inputs combined with the system outputs. A classic example is feedback being used to implement amplifier circuits.

It is possible to design feedback circuits which perform many different operations on the input signals, including addition, difference, filtering, exponential and logarithmic calculations and combinations of these and many other operations.



***An Inverting operational amplifier circuit. (OP AMP)***

***The output feeds into the input of the amplifier.***

## 2.2 Feedback in General Systems Thinking

Feedback is a concept used in general systems thinking when modeling a process, product, or production approach.

A feedback loop is a relationship between actions identified in general systems thinking. General systems thinking focuses on the way that a system's constituent parts interrelate.

An example might be to the following cycle of four activities:

1. Select an action
2. Perform the action
3. Wait for a response
4. Observe the response

As the system continues to operate the action chosen depends on the observed response. This is a feedback loop. Observation of teams has shown it is common for teams to only perform a subset of these activities. Some might even stop after the second step.

Two types of feedback are observed with general systems thinking: reinforcement and balancing.

Reinforcement is feedback that leads to an increase, or encouragement, in some element of the system.

Balancing is feedback that leads to maintain equilibrium, or diminishment, in some element of the system.

## 2.3 Feedback in Organizational Behavior

Feedback is a concept used in managing organizational behavior to provide important guidance to organizations, teams, or individuals.

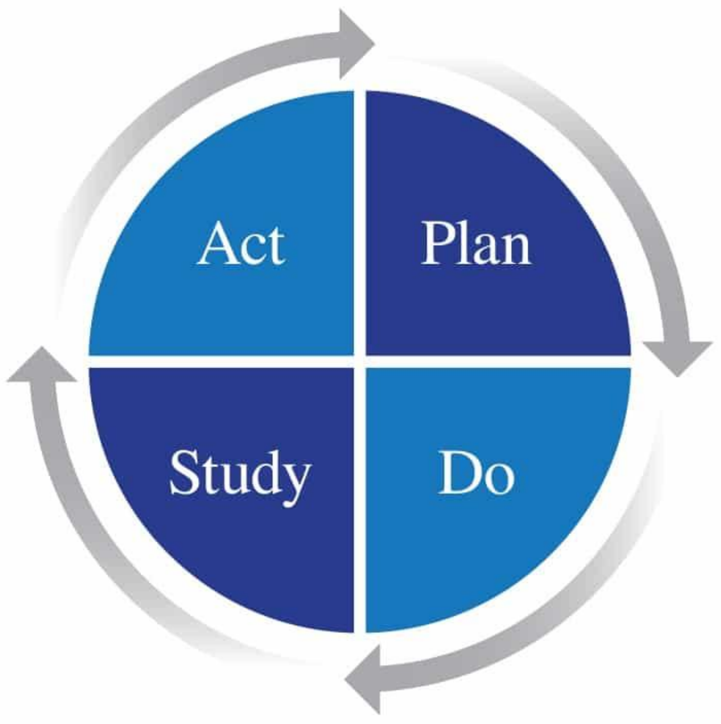
Ken Blanchard describes feedback as the "Breakfast of Champions". In his many books on the topic of organizational behavior Blanchard considers feedback as a critical success factor. When directing someone whose task is to accomplish a well-defined goal, Blanchard suggests that timely feedback should be provided to reinforce or redirect behavior. Delivering feedback is a leadership skill found in successful organizations fostering high performing teams. Feedback is part of the communication found in successful collaborative teams.

## 2.4 Feedback in Process Improvement

Edward Deming spent his career helping organizations implement quality process control systems.

Deming used a four-step feedback loop model which he called a "Shewhart" (named for his mentor), "Plan Do Check Act" or a "Plan Do Study Act" cycle.

***Deming PDSA Cycle***



Using PDSA practitioners review the results of process change to determine if it is effective. Process changes are identified to encourage improvements and to discourage degradation.

- ***Plan***, prepare for change
- ***Do***, execute the plan
- ***Study***, study the results
- ***Act***, adapt based on your findings

There are many practices which the authors have integrated into software engineering lifecycles based on the Deming Cycle feedback loop. All these practices provide the team with frequent feedback which guides quality process improvement. The team identifies behaviors to instantiate, to encourage or increase, behaviors to discourage or decrease or eliminate.

- Incremental Development
- Product Testing
- Peer Reviews
- Team Retrospectives
- Root Cause Analysis
- User Experience Surveys
- Analytics

## 2.5 Importance of Soft Skills in Feedback Loops

It seems quite easy to describe feedback from a system engineering perspective. Feedback can sound objective and clinical. The authors experience indicates that accepting or delivering feedback involves tact, diplomacy, and delicate manipulations of soft people-oriented skills. As I wrote in the book “I am a Bug!” – testing is indeed all about people and the occasional bug.

A common feedback related risk experienced by testers is in bug reports being perceived as critical of the programmer.

The authors have seen many cases in which feedback is provided to practitioners based on systematic and formal inspection models, only to find that the recipient has experienced an emotional response. The authors thought they did a great job but were shocked to discover dozens or bugs, risk of other criticisms in their good works.

Blanchard recommends practicing situational leadership when providing task related feedback to team members. The leadership style should be selected based on factors such as the level of professional experience and takes experience of the team member into account. Someone new at a task with significant experience may benefit from a more coaching style of feedback whereas a new team member working on a business-critical task may benefit from more directive feedback.

Although out of the scope of this article it is important in any feedback loops implemented as part of a software engineering process consider the soft skills and emotional responses. For bugs you might suggest that bug reports describe the behavior of the software rather than the behavior of the programmer.

The authors have found the following references to offer excellent discussions about how to develop and use soft skills in effective use of feedback loops.

- DeMarco, Tom, and Timothy R. Lister. 1987. Peopleware: productive projects and teams. New York, NY: Dorset House Pub. Co.
- Kaner, Cem and Rebecca Fiedler. 2015. Bug Advocacy: A BBST Workbook. U.S.A.: Context Driven Press.
- Gilb, Tom, Dorothy Graham, and Susannah Finzi. 1993. Software Inspection. Wokingham, England: Addison-Wesley.
- Blanchard, K. A. J. 2022. One Minute Manager (New Thorsons Classics edition). Harper.
- Seashore, Charles N., Seashore, Edith W., and Weinberg, Gerald M., 2013. The Art of Giving and Receiving Feedback, Smashwords, U.S.A.

# 3 Task Analysis of Ticket Driven Software Development

## 3.1 Critical Incident Method

The authors have used formal task analysis methods to study how software testing is implemented at different organizations, teams. The authors have used task analysis in software engineering consulting and in software engineering practices.

The critical incident method of task analysis involves studying engineering practice by interviewing practitioners about their formative experiences. Delegates are interviewed to share experiences of typical, successful, and failed practices.

Since February of 2020 Robert Sabourin performed task analysis about software testing practices at many organizations implementing variations of “agile” development with Scrum-like or Scrum-inspired development frameworks.

The organizations are in many different business domains including Enterprise Management Systems, Transactional E-commerce Systems, Medical Information Management, Financial Services, and Insurance Industries.

The common thread between these organizations was that software engineering activities were managed and tracked through ticket-oriented workflow management systems.

## 3.2 Ticket Oriented Testing

### 3.2.1 Workflow Models

In the ticket-oriented workflow systems team members invariably received work assignments as tickets which were somehow assigned to them by other team members.

Team members would receive all their work instructions through the ticket management system. If clarification was required a chain of questions were attached to the ticket. In more difficult situations the questions were exiled to enterprise chat applications or email, The problem with this is that it is too easy. for decisions to be lost.

Many tools from various vendors were observed to be used in the task analysis.

There seem to be two different approaches to completing testing work.

### 3.2.2 Testing as a Ticket Subtask

Programming, testing, data, documentation, and other task types are identified and attached to a parent ticket. The parent ticket represents a requirement or field reported bug requiring resolution. A testing task would be assigned to a testing team member.

### 3.2.3 Testing as a Ticket State

A ticket representing a requirement is set up and has many possible states. When some type of work activities is completed the state of the ticket is changed. For example, a ticket could have five states:

1. Ready for programming
2. Programming in progress
3. Ready for testing
4. Testing in progress
5. Done

When the state becomes “Ready for testing” the ticket is assigned to a member of the testing team. Ironically, in the experience of the authors, the “Ready for testing” state is usually the latest time to start testing activities related to the requirement. All of the benefits of focusing and learning the skills of incremental development are lost as a result.

### 3.2.4 Task Analysis Results

Task analysis led to several common work dynamics which the authors now use to characterize risks of using ticket-oriented workflows for managing testing activities as part of a software development lifecycle model.

Based on a survey of over 100 task analysis assessments, the following points have been frequently observed by the authors.

1. Testers have little knowledge of the relationship between their assigned ticket and the rest of development project. Often the testing work is just a stream of tickets to close rather than a prioritized list that focuses the team on learning or project risks.

2. Testers first see the ticket when it is assigned to them. They are not involved in preparing or refining the ticket.
3. Testers focus on closing ticket metrics including time and frequency. Such measures have been observed in outsourced testing projects managing work assignments via tickets. These metrics have been observed in several financial service organizations, insurance companies and contact management services primarily in distributed teams.
4. Reopening a closed ticket is looked down upon.
5. Long chains of closed form questions are used to clarify work assignment.
6. Scope and depth of testing is limited to the specifics of ticket. This tends to severely limit testing beyond the happy path. It also conditions testers away from more creative testing ideas.
7. Lack of consideration of actual technical software engineering changes implemented.
8. Lack of consideration of usage of the system to solve end user problems.
9. Lack of consideration of relevant context factors.
10. No evidence of knowledge being collected or managed other that long clarifying question chains.
11. No evidence of technical collaboration between team members is observed.
12. Test findings are used to close tickets. Test findings are not used to improve the software development process model.
13. Reduced understanding of the larger architectural frame of the project. Everything is seen through a keyhole so the whole is a mystery.

There is little evidence of the use of feedback loops in ticket driven software engineering workflows. The only feedback loop observed were “agile” retrospective rituals mentioned by some delegates. Maybe some metrics around team performance are capture (cycle time).

# 4 Feedback Loops with Product Owners

## 4.1 Product Owner Role

On a software engineering team, the product owner role is that of a customer advocate. Product owners are responsible for understanding and communicating software requirements to the other team members.

Product owners are responsible for prioritizing requirements based on business needs and in collaboration with the development team.

Product owners are responsible for establishing what quality means to the team and ensure that this relates to business goals and customer expectations.

Product owners have budget responsibility (most often by controlling for project duration) for the software development project.

A feedback loop between a tester and the product owner can both guide testing and drive decision making. This is one of the most active feedback loops as feedback from testers in the best case can help drive ready for release decisions if the product owner is tuned to this information. In the worst-case testers are frustrated as product owners release despite the testers concerns. It is important for the product owner to make clear the business drivers that often drive these otherwise puzzling decisions.

## 4.2 Context Factors

Context factors are variables which influence how decisions are made in a project. Testers can scope testing activities based on knowledge of context factors. The scope of testing describes which elements of a testable object should be exercised and which elements of a testable object can be excluded.

Active context listening is a feedback loop between the product ownership role and a testing role.

Here are some example steps in the active context listening feedback loop.

1. Product owner makes a prioritization decision
2. Tester identifies, and isolates the context factors driving the prioritization decision
    a. Business
    b. Technical
    c. Organizational
    d. Cultural
    e. Quality
3. Tester categories identified context factors
    a. Are there and new context factors?
    b. Are the context factors already known?
    c. Did the impact of a known context factor change?
4. Tester updates their scoping model based on context factors
5. Any in process testing is adapted based on changes to test scoping model.

The tester by monitoring prioritization decisions of the product owner role can adapt, support, or redirect test scoping.

Note that in this context listening approach all team members including the product owner have visibility into how testers define the scope of a testing activity.

### 4.3 Product Requirement Feedback loops

Testers have opportunities to influence the product requirements development teams are called upon to implement. Requirement management approaches vary dramatically across different software development lifecycle models, technologies, and domains. Some lifecycle models have dedicated rituals enabling teams to refine requirement definitions. Some lifecycle models rely on formal reviews or inspections to improve or adapt the statement of product requirements.

The basic feedback loop between testing roles and product owner roles follows an iterative pattern.

1. Product owner updates requirement statement or product description
2. Team members review requirement statement
3. Team comes to consensus on requirement critique
    a. Relate requirement to other aspects of product
    b. Disambiguate statement of requirement
    c. Identify missing attributes of requirement
    d. Identify unnecessary elements of requirement
    e. Improve requirement acceptance criteria
    f. Estimate work required to implement requirement

# 5 Feedback Loops with Programmers

Some testers frequently interact with programmers. Feedback loops can be established between programmers and testers enabling faster deliberately focused testing and minimizing development rework.

### 5.1 Design and Planning

During planning, programmers establish a technical direction to solve the problem of implementing a product requirement, bug fix or change request.

Although often the programming approach is governed by technical intricacies indicated by tools, frameworks, platforms, operational concerns, and software architecture, there is still an opportunity for a direct feedback loop between testers and programmers related to design.

Consider the following design approached used by a customer of Robert Sabourin during the teams Scrum planning session.

1. Programmer identifies two or more design alternatives
2. Programmer models solution indicating scope and nature of technical changes
3. Programmers identify benefits associated with each alternative
4. Testers identify potential risks associated with each alternative
5. Action take
    a. Suitable alternative selected
    b. New alternative added
    c. Existing alternative adapted

Even though testers may not be skilled in software design and architecture (though the exceptions are less rare than you might think), they are able to participate in an affinity analysis of the design alternatives considering the benefits and consequences of each alternative. Essentially understanding what can go wrong or how resilient a particular approach is can help the team chose the best course of action.

Testers can be key at this stage to see issues in things like state models, complex business rules, operational issues and the like that can be very valuable. Testers can be involved in the evaluation of proof-of-concept spikes to evaluate design alternatives which is another feedback input that can be very valuable.

The design decision feedback loop allows testers to provide critical input establishing the technical approach taken by the team.

## 5.2 Understanding Code Changes

While consulting in the desktop security domain Robert Sabourin established programmer tester feedback loops related to understanding code changes in order to focus regression testing. The regression testing was expected to expose unintended side effects of code changes.

The feedback loop was implemented with the following steps.

1. Tester reviewed code changes with programmer
2. Each changed module was assigned a risk factor based on the nature of the technical change
    a. No change in program logic
    b. Some change in program logic
    c. Significant change in program logic
    d. Redesign program logic
3. Tester identified usage scenarios related to changed code
4. Action
    a. Tests revealed bugs requiring attention
    b. Tests did not reveal bugs requiring attention

In this case testers understood the relationship between source code module and workflow of the software under development.

When bugs were revealed, programmers completed rework and instantiated another iteration of the feedback loop.

## 5.3 Paired Testing in the Development Environment

While working in the medical software industry it was a common practice for testers and programmers to participate in a type of exploratory testing in the development environment before the code was committed to the main code line. The testing took place on the developer’s desktop using the day-by-day development environment including a private data set completely in the control of the programmer

The charter for this exploration was twofold.  The tester had a chance to become familiar with new behaviors being implemented in the system.  The developer had a chance to exercise the software while it was still a work in process thus making it relatively straight forward to tweak the systems behavior without the need of an additional build cycle,

Here are the basic steps involved in the paired exploratory testing feedback loop.

1. Programmer implements requirement to team definition of done, thus completing unit testing and any technical reviews or other established checklist items
2. Tester identifies usage scenarios related to newly implemented requirements
3. Tester identified variables impacting or impacted by the identified usage scenarios
4. Tester uses test design techniques to select values for variables of interest
5. Programmer sets up development environments with break points and logging of code being exercised.
6. Tester and programmer walk through usage scenario with prescribed values for selected variables observing systems behaviour and ensuring code is executed as intended.
7. Action
    a. Bugs are identified, and attended to on the spot
    b. Tests do not reveal bugs
    c. Tester learns about the newly implemented system behaviour
    d. Tester identifies technical risks based on code changes
    e. Tester identifies strategies to assess correctness
    f. Tester identifies mechanisms such as hooks to facilitate testing and result interpretation
    g. Tester identified testing ideas based on the technical changes to the source code

## 5.4 Bug Reporting

Bug reports are a potentially influential deliverable of a testing activity.

Many software testing texts, articles and courses have been dedicated to guiding testers in excellent bug reporting.  Cem Kaner's book "Bug Advocacy: A BBST Workbook" offers respected guidance to writing an effective bug report.

Bug reporting is a part of two important feedback loops in software engineering,

### 5.4.1 Bug Improvement

The tester actively elicits feedback from peers, programmers and other stakeholders about the form and content of the bug report.  Based on feedback the tester improves the statement of the bug including information which helps teams make better decisions about bugs, helps programmers isolate and correct the bug efficiently and helps support organizations help customers work around or otherwise deal with the potential impact of the bug.

Improvements in bug reporting usually result in a combination of improved individual skills and standard practices and guidelines for future defect reporting and management.

### 5.4.2 Bug Triage

#### 5.4.2.1 Feedback in Triage

Invariably software development teams need to decide what to do about a bug.  Do we fix it now?  Should we fix it later?  Should we leave it in the product?  Should we publish a way to work around the problem?

When working in the medical knowledge base field, Robert Sabourin established some important tester, programmer feedback loops relating to bug triage, A helpful heuristic had to do with the effectiveness of testers.  During projects the team established that the most effective testers were testers who found and reported bugs that ended up being fixed.  Effective testers were able to persuasively advocate for their bugs. Effective testers learned which types of bugs to look for and did not invest a lot of effort in searching for bugs that would not get corrected.

The basic feedback loop was described as:

1. Tester identifies a bug
2. Tester reports the bug
3. Bug is triaged
4. If bug is corrected
    a. Tester learns factors making this bug important
    b. Encourage testing which would expose bugs with similar factors
5. If bug is not corrected
    a. Tester learns factors making this bug unimportant
    b. Discourage testing which would expose bugs with similar factors

Note that any bug identified was always reported, what changed was the type of bug testers were looking for. For example, if grammar errors were not considered important then testers would not explicitly look for grammar errors, however if the tester chanced upon a grammar error it would be reported.

#### 5.4.2.2 Becoming an effective Bug Advocate

An effective tester is an informed tester sensitive to context and skilled in rhetoric.

Feedback loops are critical to help testers understand the impact of context on both the severity and priority of bugs.

Robert Sabourin, when working in the medical software domain, observed many bugs that changed both priority and severity based on project context. A clinician could be distracted by a subtle cosmetic error. The same clinician would never trust software which conspicuously crashed. Counterintuitively, the conspicuous crash could have a lower severity than the cosmetic error. The conference presentation, “The Elevator Parable” [33] shows several examples of priority and severity being sensitive to context.

The effective tester learns from stakeholder feedback about the relative importance, and risks, associated with bugs. The effective tester judiciously chooses which bugs to advocate.

Feedback loops are also critical in helping the testers understand the value and limits of persistent argumentation about bugs. As Ross Collard indicates in “The Tester’s Guide to People & Organization Issues the Tester’s Survival Guide” [31], the successful tester is a great salesperson. The skills of persuasion needed to convince stakeholders that a bug should be fixed are familiar to expert sales professionals. Many sales skills are about relationships and values. Testers should look at books such as [32] Johnson, Spencer, and Larry Wilson. 2002. The One Minute $Ales Person to pick up some tips about sales skills that matter.

## 5.5 Troubleshooting and Debugging

### 5.5.1 Troubleshooting

A lot of training focuses testers on the act of identifying and reporting about bugs in software. Rich terminology and vocabulary are established to emphasis the difference between errors, failures, issues, incidents, and defects.

The authors of this paper come from an engineering background. The authors consider it critical for testers to provide insights into the cause of a problem, not just the externally observed behavior of the system. Testers often have deep knowledge of the system data for example that can be invaluable in understanding many types of problems.

The authors define troubleshooting as a term used to describe isolating the cause of a specific problem.

This is an example of a tester programmer feedback loop involving trouble shooting.

1. Programmer implements a requirement
2. Tester identifies a requirement related unexpected behavior of the system under test
3. Tester formulates a hypothesis of which factors drive the unexpected behavior
4. Tester runs a trial experiment to confirm or refute the hypotheses
    a. If not confirmed, consider another factor
    b. If confirmed report bug

#### 5.5.2 Debugging

The authors define debugging as is a series of technical activities directed at finding ways to resolve a problem.

In the text, Systematic Software Testing, SST, author Rick Craig indicates that testing and debugging should be isolated from each other.  In SST, testers are expected to identify defects but are not expected to isolate their cause nor to find a solution.  In SST, testing is not debugging.  In SST, testing is not troubleshooting.

This is an example of a tester programmer feedback loop involving debugging.

1. Tester reports a bug
2. Programmer formulates a hypothesis of which factors influence the bug
3. Programmer runs a trial experiment to confirm or refute the hypotheses
    a. If not confirmed, consider another factor
    b. If confirmed, then correct the bug

#### 5.5.3 Active participation by Testers

A tester can work with a programmer in the development environment to help identify which factors cause a problem and to verify that a proposed solution yields the correct results.

Testers can pair with programmers and other team members to go beyond just identifying a problem, they can collaborate to help resolve the problem as well. You do not need to be a programmer to learn bug isolation and debugging techniques with a modern development environment.

# 6 Feedback Loops Driving Process Improvements

## 6.1 Bug Clusters

Many aspects of software testing can influence process improvement.  Testing findings can help identify quality concerns about a product.  Test findings can also help identify potential process improvements for all aspects of software development.

When testers identify multiple bugs which have a common cause then a bug cluster has been identified. By eliminating the cause multiple bugs can be corrected and future related bugs can be avoided.

In order to understand the cause of bugs testers should review with programmers the actual technical work done to correct the defect.  Clusters are defined by common cause, not necessarily by common effect.

## 6.2 Team Retrospectives

Team retrospectives are examples of feedback loops directed at the work dynamic of the entire team.

Many “agile” teams hold a retrospective meeting immediately following each development iteration or sprint. The retrospective meeting reviews the work done in the sprint.

The goal of the retrospective meeting is to review work done by the team in the previous sprint and look for opportunities to self-organize encouraging effective practices and changing ineffective practices. Changes are to be implemented in the following sprint.

During the retrospective teams review the technical work done. The team discusses the following points:

- What worked well?
- What did not work well?

The team notes practices to continue using and reinforce.

For points that did not work well the team discusses alternatives and decides upon changes redirecting their method of operation for the subsequent sprints.

The retrospective meeting is used to identify improvements in programming and testing tasks. The team discusses the following points:

- Any wasted time?
- Any missing tasks?

The team uses knowledge gained to improve planning in future sprints.

If a task was a waste of time then the team discusses whether it could be avoided or implemented differently in the future. Missing tasks could be added to checklists maintained to help planning future sprints.

The team reviews collaboration within the team and with people outside of the team. Excellent collaboration practices are to be encouraged. Weak collaboration practices lead to discussions of how to improve and redirect such collaboration in the future.

# 7 Feedback Loops with the User Community

When working under extreme time pressure the authors have had to make some important tradeoffs in testing. Deciding which testing activities deserve more attention, or more effort, can help teams manage product risks.

Robert Sabourin had recent experience in a medical analysis company to identify testing focus under strict time pressure working to a fixed deliver date release. In this project Robert set up a feedback loop between the user community and the testing team. Note that even though the system under test had hundreds of menus, controls and dialogues, there were only 30 usage scenarios identified.

The intent was to learn what users did and thus to focus testing on ensuring the users could do their job with the system under test.

Several short task analysis interviews took place to identify who were the users and subsequently what did they do.

The idea was to model what the users did with the system instead of the traditional test model of what the system does for the user. Respectfully rephrasing John Kennedy’s inaugural address “Ask not what the software does for the user but ask what the user does with the software”.

The basic elements of the user tester feedback loop were:

1. Identify different types of users
2. For each user type identify the tasks they are trying to accomplish
3. Prioritize usage tasks
    a. Pareto analysis
    b. Business importance
    c. Technical risk
4. For each high priority task
    a. Identify factors which influence outcome of task
        i. Conditions
        ii. Variables
        iii. Factors
        iv. Environment
        v. Coresident software
    b. Identify factors which are influenced by outcome of task
    c. Select values for factors
    d. Walk through scenario from start to end
        i. Pair with member of user community
    e. Identify concerns
        i. unexpected behaviors
        ii. inconsistent behaviors
        iii. missing behaviors

# 8 Feedback Loops in Session Based Exploratory Testing

## 8.1 Exploratory Testing

To begin with the authors could suggest that all testing is exploratory and thus exploratory testing is just another word for testing.

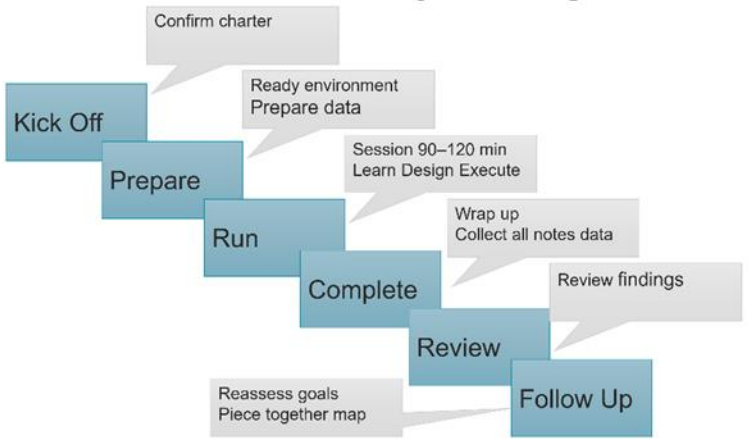
Cem Kaner posted the following description of exploratory testing on his blog entitled: “On the craft and community of software testing”.

> “Exploratory software testing is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.”

## 8.2 Charter Driven Exploratory Test Feedback Loop

An example exploratory testing framework implements the following steps in a feedback loop:

Charter Driven Exploratory Test Framework



The tester will be exploring. A collaborator will be commissioning the exploration. The collaborator can be a programmer as is more common in an "agile" team. The collaborator can be a test-lead as is more common in a traditional or structured software development life cycle such as Waterfall, Rational Unified Process or V-Models.

| Step | Description |
|---|---|
| Kick Off | The tester reviews the charter with the person commissioning the testing. The tester makes sure that the scope and depth of the charter are discussed and agreed to. The duration of the upcoming session is established. Testers should review which variables, factors, conditions, or data sources may be relevant. |
| Preparation | The tester gathers whatever resource, data, tools, or equipment are required for the upcoming session of testing. |
| Run | This is a time boxed session. The tester is expected to focus 100% of their attention of the session of testing. The tester will design and execute tests trying to learn about the charter. The tester will keep a record of decisions made, of tests attempted and of observations. The |

| | |
|---|---|
| | tester will use many diverse tools and technologies to complete this step. |
| Completion | The tester gathers their findings. The tester reports any bugs in bug tracking tools as required by the project team's workflow. The tester relinquishes the environment. Note that the tester must be able to reset the environment to a predictable state so very often at the completion step the testing will create virtual images of the test environment and data. |
| Review | The tester reviews their findings with the person who commissioned the testing. In "agile" teams this is often the developer. In the review meeting all findings are reviewed and decisions are made on how to act on the finding in essence the review step is culling test findings and turning them into action. Test results are fed back to the person who commissioned the testing and to any stakeholder would benefit from knowledge of the findings. This is a call for action. The review meeting usually takes place on the same day as the session of testing.<br><br>The key decision made is – should we do another session on the same charter or should we move onto something else. |
| Follow Up | The team acts based on the finding of the tester.<br><br>The tester acts based on the feedback from the person who commissioned the work and any other stakeholder involved. This list will vary from charter to charter. |

## 8.3 Charters

A test charter is a mission statement for testing. It is a goal. What does the tester what to learn about?

Note that charters derive from test ideas. One test idea can map to many charters. Multiple test ideas can map to one charter or there can also be a one-to-one mapping between test ideas and test charters.

On an "agile" team a charter statement can be the "name" or "title" of a testing task.

## 8.4 Session Based

Session based testing is implemented one time-boxed session at a time. Sessions are typically 90 minutes to 2 hours long. Sessions can be shorter or longer, but it is important to agree on the session duration before starting to test.

During a testing session the tester is uninterrupted. The tester focuses on designing and executing tests related to the charter. Testers keep track of their work and record their findings.

At the end of a session the tester reviews their findings with the charter commissioner who is typically a programmer or a test lead depending on the lifecycle model being used.

When reviewing the findings, a decision is made as to whether additional sessions should be implemented to further fulfill the test charter.

## 8.5 Test Findings

There are many ways that exploratory testers can express their findings. A charter can be represented by a task in a workflow management system, for example a Jira ticket. Each session associated with that charter would need to be a sibling object. In a session object the tester would include their findings, session notes, screen shots, screen videos with audio commentary, spreadsheets, data records, virtual images of system under test, pointers to bug descriptions and commentary from developers, product owners, teammates, and other interested project stakeholders. Collecting and recording findings should be a natural part of the testing workflow.

Session notes are like medical notes on a patient chart or a professional engineer logbook. This is a record of the testing done describing decisions made and trials attempted. The session notes do not need to include analysis or assessment, generally session notes focus on recording facts. Some team cultures allow for a section of analysis and recommendations from the tester.

Many testers choose mind mapping tools such as XMind or FreeMind to capture visual representation of test findings. Mind maps can include images, text, links to other objects and relationships between objects.

When testing in a regulated environment consistent session note can be used to demonstrate compliance to regulatory standards.

## 8.6 Feedback Loops to the Programmer

The programmer acts on findings from the session review. This may lead to code modification, bug fixes or further experimentation with the technical solution being implemented.

## 8.7 Feedback Loops to the Tester

The tester learns how to focus their work. Were the findings relevant? Was the scope inclusive of factors of importance? Were some factors superfluous were other factors on target? Were the bugs identified relevant to the project?

The tester adjusts the scope and focus of testing,

The tester adjusts the variables under study.

The tester learns how test findings are used and interpreted. This feedback is important to adjust the information gathered and how it is both recorded and reported. The tester streamlines their notes to focus on actionable information without distraction.

### 8.8 Feedback Loops to and from Other Stakeholders

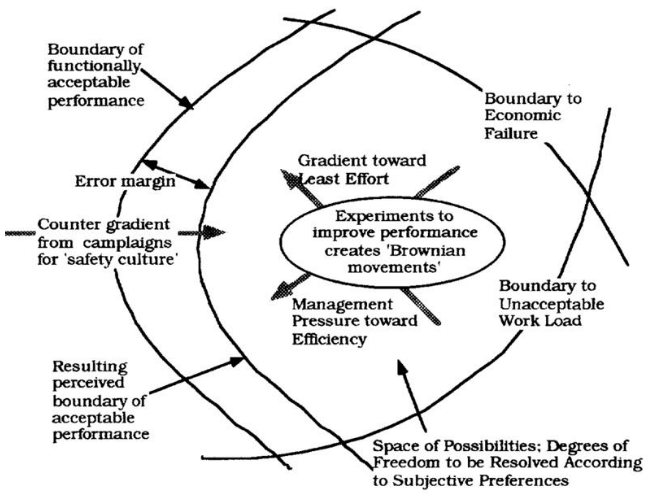
Exploratory test findings can be reviewed in feedback loop with many other project stakeholders. Here are some examples from recent projects.

- The "***customer***" provides feedback about whether the product delivers the value required.
- An "***end user***" provides feedback about whether they can complete tasks and workflow with the software under test.
- A "**domain expert**" provides feedback on strategies to assess correctness and functionality omissions.

# 9 Feedback to Management and the Business

Testers can also provide feedback directly to the business. As mentioned earlier the feedback loop between the product owner and the testers is an active and critical feedback loop for projects. A less talked about feedback loop is direct to management and "the business". We often talk about "what the business wants", high level roadmaps over several years (whether you think these are valid, most businesses have them), and other context drivers that make this particular feedback loop so interesting.

One example is a company that needs to demonstrate a particular feature set or capability to a key customer or investors. This is common for companies in the startup phase or ones trying to break into a new market. This demo becomes everything for the team and has a very different focus than normal product development. The team needs to be careful to pay attention to non-functional requirements so they don't regret creating a system they can't operate effectively, but these concerns are diminished in favor of the creation of an effective demo. In our experience management is often very involved in the demos and lean on the testers as much as the developers in understanding if the demo is ready. It isn't uncommon for the testers to be very involved in the creation of the demo script. It's an art to have a demo that is effective but doesn't imply more than the team can deliver.


Boundary of functionally acceptable performance
Boundary to Economic Failure
Gradient toward Least Effort
Error margin
Counter gradient from campaigns for 'safety culture'
Experiments to improve performance creates 'Brownian movements'
Boundary to Unacceptable Work Load
Management Pressure toward Efficiency
Resulting perceived boundary of acceptable performance
Space of Possibilities; Degrees of Freedom to be Resolved According to Subjective Preferences

Jens Rasmussen in "Risk Management in a Dynamic Society" has a fantastic summary of what drives systems, which of course includes software. This is another "triangle" in software development that everyone feels implicitly and seems to be gaining more explicit attention with the increase in systems thinking and complexity that is seeping into our industry. It's easy to see where the concerns that dominate what the business is thinking about, and the concerns of the team intersect. This balance is well informed by testers who tend to be closer to the business and customers than the rest of the team. As all the forces in play are always pulling at each other, testers are often key to finding balance.

- What issues have workarounds that the customer can live with (helping to ship on time)?
- Are there re-orderings of workflows that can make performance acceptable?
- What bugs just must be fixed? (Harking back to the bug advocacy discussion and concerns about support costs)

Management feels like a constant risk management problem that needs to balance the resources that are available (budget, time, vendors, technology, team skillset, etc.), c-level and investor expectations, the energy of the team, customer expectations and all the rest. Testers are often in the intersection of these concerns and play an important part in getting the best outcome.

## 10 Concluding Remarks

Feedback loops can help amplify the effectiveness of testing and the overall software development process.

Testing related feedback loops are demonstrated to:

- improve product requirements early, before implementation begins
- influence software design decisions
- improve personal and team software process
- Increase engagement in active processes instead of milestones and phase gates that loom over the team.
- to dynamically adapt the focus, scope and depth of testing based on active feedback
- to influence process, change in self-organized teams to improve product quality
- find and fix the bugs that matter sooner
- provide vehicles to collaborate with stakeholders, programmers, and members of the user community

## 11 Acknowledgments

The authors wish to thank their peers in the software engineering community.

Robert Sabourin would like to thank the thousands of students who have participated in his software engineering and software testing courses over the past several decades.

## References

[1] Myers, et al. The Art of Software Testing. John Wiley & Sons, 2012.

[2] Kaner, Cem, and James Bach. Lessons Learned in Software Testing. Wiley, 2001

[3] Pólya, George. How to Solve It: A New Aspect of Mathematical Method. Doubleday, 1957.

[4] Sommerville, Ian. Engineering Software Products. Pearson Education, Inc., 2020.

[5] Copeland, Lee. A Practitioner's Guide to Software Test Design. Artech House, 2008.

[6] Sabourin, R. Charting the Course Coming Up with Great Test Ideas Just in Time. AmiBug, 2020.

[7] Dijkstra, Edsger, “Programming methodologies, their objectives and their nature.” 1969

[8] Hersey, Paul, and Kenneth H Blanchard. 1969. Management of Organizational Behavior: Utilizing Human Resources. Englewood Cliffs, N.J: Prentice-Hall.

[9] Seashore, Charles N., Seashore, Edith W., and Weinberg, Gerald M., 2013. The Art of Giving and Receiving Feedback, Smashwords, U.S.A.

[10] Blanchard, Kenneth H, Patricia Zigarmi, and Drea Zigarmi. 1985. Leadership and the One Minute Manager: Increasing Effectiveness through Situational Leadership. 1st ed. New York: Morrow.

[11] Blanchard, K. A. J. 2022. One Minute Manager (New Thorsons Classics edition). Harper.

[12] Shewhart, Walter A. 1939. Statistical Method from the Viewpoint of Quality Control. Edited by W. Edwards Deming. Washington: Graduate School, the Dept. of Agriculture.

[13] Deming, W. Edwards, and Joyce Nilsson Orsini. 2013. The Essential Deming: Leadership Principles from the Father of Total Quality Management. New York: McGraw-Hill.

[14] Kiran, D. R. 2016. Total Quality Management: Key Concepts and Case Studies. U.S.A.: Elsevier Ltd.

[15] Jonassen, David H, Martin Tessmer, and Wallace H Hannum. 1999. Task Analysis Methods for Instructional Design. Mahwah, N.J.: L. Erlbaum Associates.

[16] Kaner, Cem and Rebecca Fiedler. 2015. Bug Advocacy: A BBST Workbook. U.S.A.: Context Driven Press.

[17] Geyer Studio, Copyright Claimant. Inaugural address, by John Fitzgerald Kennedy, President of the United States, 1961 to 1963., ca. 1966. Photograph. https://www.loc.gov/item/2015649386/.

[18] Craig, Rick D, and Stefan P Jaskiel. 2002. Systematic Software Testing. Artech House Computing Library. Boston: Artech House.

[19] Weinberg, Gerald M. 2001. An Introduction to General Systems Thinking Silver anniversary ed. New York: Dorset House.

[20] Cohn, Mike. 2004. User Stories Applied: For Agile Software Development. Addison-Wesley Signature Series. Boston: Addison-Wesley.

[21] Derby, Esther, and Diana Larsen. 2006. Agile Retrospectives: Making Good Teams Great. The Pragmatic Programmers. Raleigh, NC: Pragmatic Bookshelf.

[22] Schwaber, Ken, and Mike Beedle. 2002. Agile Software Development with Scrum. Series in Agile Software Development. Upper Saddle River, NJ: Prentice Hall.

[23] Schwaber, Ken. 2004. Agile Project Management with Scrum. Redmond, Wash.: Microsoft Press,

[24] Freedman, Daniel P, and Gerald M Weinberg. 1990. Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products. 3rd ed. Little, Brown Computer Systems Series. New York, NY: Dorset House Pub.

[25] Gilb, Tom, Dorothy Graham, and Susannah Finzi. 1993. Software Inspection. Wokingham, England: Addison-Wesley.

[26] Wiegers, Karl Eugene. 2002. Peer Reviews in Software: A Practical Guide. The Addison-Wesley Information Technology Series. Boston, MA: Addison-Wesley.

[27] Fagan Michael, “A History of Software Inspections” in Broy, M, Ernst Denert, and Sd & m AG. 2002. Software Pioneers: Contributions to Software Engineering. Berlin: Springer.

[28] Kaner, Sam; Lind, Lenny; Toldi, Catherine; Fisk, Sarah; Berger, Duane; Doyle, Michael. 2007. Facilitator's Guide to Participatory Decision-Making. Hoboken, NJ: Jossey-Bass.

[29] DeMarco, Tom, and Timothy R. Lister. 1987. Peopleware: productive projects and teams. New York, NY: Dorset House Pub. Co.

[30] Kaner, Cem, 2006, “Defining Exploratory Testing”, https://kaner.com/?p=46

[31] Collard, Ross. The Tester’s Guide to People & Organization Issues the Tester’s Survival Guide. New York, Collard & Company, 2008.

[32] Johnson, Spencer, and Larry Wilson. 2002. The One Minute $Ales Person. New York: W. Morrow.

[33] Sabourin, Robert. 2003. "Establishing Bug Priority And Severity: The Elevator Parable". Stickyminds. https://www.stickyminds.com/presentation/establishing-bug-priority-and-severity-elevator-parable.

[34] Rasmussen, Jens. 1997. “Risk Management in a Dynamic Society: A Modelling Problem.” Safety Science Vol. 27, No. 2/3. pp. 183-213.