WHITE PAPER

Retrofitting Legacy Systems with Unit Tests

Jenny Stuart, VP Consulting, Construx Software
Melvin Perez, Senior Fellow, Construx Software

Version 1.5 4, July 2018

Automated unit testing improves organizational efficiency. The sooner defects are found, the more efficiently and effectively developers can fix them. However, many companies have existing systems with little to no unit tests, and retrofitting those systems completely with comprehensive unit tests is impractical. The time and effort required to perform a complete retrofit make it difficult to justify and obscures how the organization can get a positive return on investment with unit testing. This white paper discusses how to implement unit testing on a legacy system.

This white paper was downloaded from
www.construx.com/whitepapers

Construx

Copyright

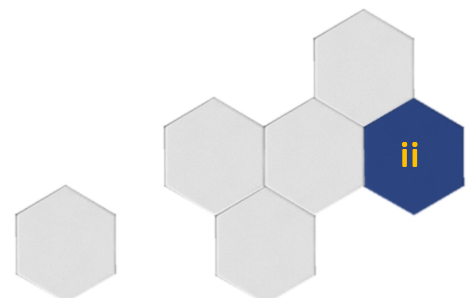
© 2011-2018, Construx Software Builders, Inc. All rights reserved.

Construx Software Builders, Inc.
10900 NE 8th Street, Suite 1350
Bellevue, WA 98004
U.S.A.

This white paper may be reproduced and redistributed as long as it is reproduced and redistributed in its entirety, including this copyright notice.

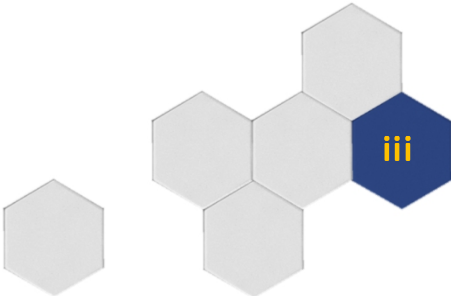
Construx and Construx Software, are trademarks of Construx Software Builders, Inc. in the United States, other countries, or both.

This paper is for informational purposes only. This document is provided "As Is" with no warranties whatsoever, including any warranty of merchantability, noninfringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Construx Software disclaims all liability relating to use of information in this paper.



Contents

Reasons to Retrofit 1
Decide the Approach to Building Out Unit Testing 2
Create the Necessary Infrastructure 4
Build the Tests 6
Incrementally Improve System Testability 8
Contributors 11



Reasons to Retrofit

Without automated tests, it does not matter how well designed a system is—there will be concern about and risks to changing or extending it. This is because there is not an efficient way to verify that the system is still working and whether it is being improved or degraded after every major change.

Legacy systems cannot remain unchanged, because the business they support does not remain the same over time. Common changes include adding new features, fixing defects, improving the design, optimizing resource usage, and making the system compliant with a specific regulation or standard. An organization considering retrofitting automated tests is looking for an effective way to minimize the risk of breaking the system when these changes are introduced.

The first aspect to consider is the importance/risk of each system. It is common to vary the strategy for unit testing based on aspects outlined in the following table. These aspects can be used to create a “risk profile” for each system.

Table 1

Area	Ranking Description
Consequence of error	1=Nuisance only; 5=Risk to human safety
Percentage of clients using the software	1=Internal use only; 5=Used by 80%+ of clients
Amount of current/planned development	1=No active development or little active development; 5=New product offering
Level of stability/brittleness	1=Easy to modify without unexpected consequences; 5=Brittle and hard to maintain
Error history	1=Limited field failure reports; 5=Numerous field failure reports

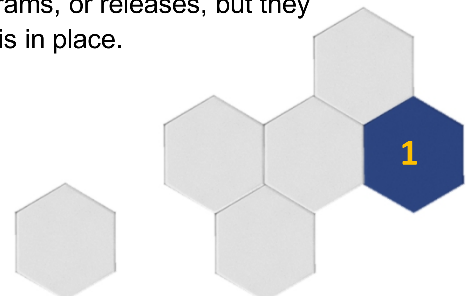
Score of:

5-10 No investment or limited investment is recommended.

11-18 Some investment is recommended.

19+ Significant investment is recommended.

When determining the amount of the investment in retrofitting automated tests, take into consideration the opportunity cost. What else could the organization do with the estimated investment? Most organizations find that adding automated unit testing initially adds work to their projects, programs, or releases, but they see major dividends once a meaningful set of tests is in place.



Decide the Approach to Building Out Unit Testing

Construx cautions its clients about trying to completely retrofit existing code bases with automated unit tests. This is tedious work with productivity costs that are far too prohibitive for most organizations. Instead, organizations have been successful with incrementally increasing automated unit-test code coverage.

Select strategies for incrementally building out coverage

The strategy for retrofitting systems varies from organization to organization, as well as across the overall portfolio. It is important to evaluate the current portfolio and make decisions about where and how much to invest in this effort.

The ***most common strategy used in companies is to add unit tests whenever code is modified:*** when new code is added, when defects are fixed, and/or when code is changed for maintenance. Other strategies include:

- To areas more likely to change. Analyze change history and develop tests for those areas with high change rates.
- For difficult-to-work-on or error-prone areas of the system. For example, profile the system and then develop tests for components that have high defect counts or that have unusually high complexity metrics.
- For critical intellectual property. For example, test algorithms that are essential to the business domain.
- In reusable code. Identify the components with high fan-in or afferent coupling metrics and develop automated tests for common usage scenarios.

An example of a risk profile and the selected approaches for testing it are outlined in the following table.

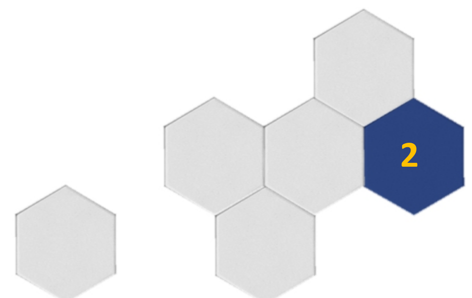


Table 2 Retrofitting Strategy Example

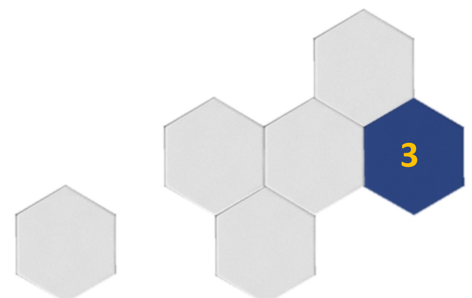
System	Investment Level	Unit Test Retrofitting Strategy
A	None	No investment will be made in unit testing.
B	Medium	Unit tests will be developed for the highly complex and defect-prone areas of the system. Unit tests will be added for new code, all defect fixes, and maintenance work.
C	High	Unit tests will be developed for the highly complex and defect-prone areas of the system. Unit tests will be added for new code, all defect fixes, and maintenance work. Unit tests will be developed for core business algorithms.
D	Medium	Unit tests will be developed for the highly complex and defect-prone areas of the system. Unit tests will be added for new code, all defect fixes, and maintenance work.
E	Low	Unit tests will be developed for the highly complex and defect-prone areas of the system.

Consider establishing goals for test coverage

Construx recommends that, as unit testing is added, organizations consider establishing code-coverage goals and using tools to evaluate the current coverage.

However, keep in mind that achieving 100% coverage in a legacy system is typically unachievable and/or does not have a high ROI. Instead, the coverage goals should be linked to the selected strategy and used to understand if that is being achieved.

A coverage goal might look more like “x% coverage of all new code, y% of the code selected for retrofitting of unit tests, and z number of refactored modules.” For example, in a legacy system with no existing coverage, the goal may be to get 5% coverage in all areas of the system and 80% coverage of a set of core algorithms within 18 months.



Create the Necessary Infrastructure

Invest resources

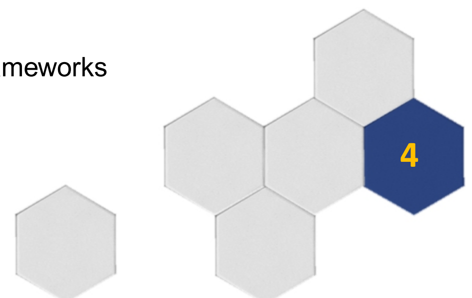
Establishing unit testing infrastructure is work, and the organization needs to allocate staff and funds for this work. Staff need time to evaluate unit-test and coverage tools, build unit-test infrastructure, integrate the tools into the build system(s), and help teams to begin to use the infrastructure. To be successful, the organization needs to establish a team of people who will have time to participate in this work. This “unit test adoption team” should include people who work in different systems, technologies, and languages.

Select unit-testing tools

Establishing the unit-testing infrastructure includes understanding the language/technology mix in the organization, identifying candidate tools, evaluating the tools, and selecting a tool set. In most cases, organizations with legacy systems also have a mix of technologies. This means that establishing a single standard is impractical. The same unit-testing tools will not work for C#, C++, Java, COBOL, etc. Fortunately, there are xUnit frameworks available for virtually all the major programming languages[1].

Rather than looking for one tool that fits all its needs, organizations should establish a set of tools that supports its technology mix. The first step is to inventory all the languages/technologies in use and determine which unit test (and coverage) tools are good candidates. The second step is to evaluate the resulting candidate tools against a set of criteria, including using the tools on some of the organization’s actual systems. Beyond the selected tools, many organizations need to consider strategies to create and manage mocks, stubs, and dependency injection. Most legacy systems require the creation of these to support effective unit testing.

[1] http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks



As unit testing is introduced, organizations might need to consider the legacy code dilemma[2]. Although retrofitting tests enables the organization to safely introduce changes in the future, putting tests in place often requires changes to the code itself. These changes are mainly needed to break dependencies so that testing is easier and/or possible. Construx recommends having a secondary testing mechanism in place to verify that the changes introduced by adding tests are not breaking the system. For example, a "user robot" mechanism could be used to record specific *characterization tests* via the user interface that could be played back as changes are introduced. Depending on the risk profile, this kind of tool could be required.

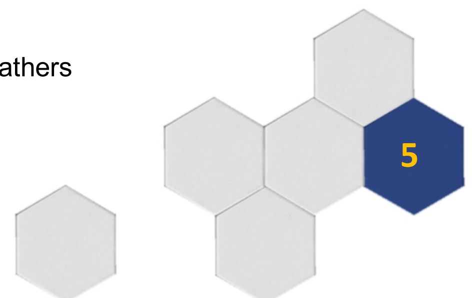
Integrate unit tests with the build system

As the automated unit-test framework is established, the organization must be sure to integrate it with the build process. The inclusion of these test suites ensures that the core system functionality maintains a baseline level of quality. Without this, test builds merely ensure the system builds correctly. It is possible for a system that builds correctly to fail upon first execution or to have issues in major functional elements.

The unit tests (or a subset of them) should be automatically executed after each build, the developers should be able to run the test suite in their local environments, and the build engineer should be able to run it on demand. Many organizations also select a code-coverage tool and run it as part of a nightly build.

---

[2] *Working Effectively with Legacy Code*, Michael Feathers



Build the Tests

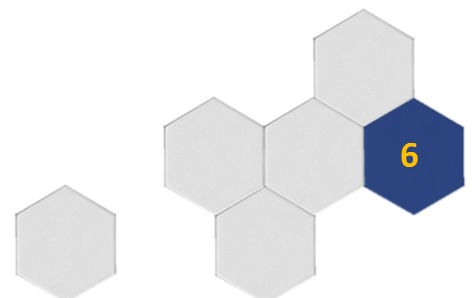
After the automated unit-testing tools and strategy for implementing unit testing have been decided upon, the organization is ready to begin retrofitting its system(s). At this point, the main question is “What is needed to help the teams begin building useful unit tests?”

Train staff to perform unit testing

Effective unit testing improves the design of the software. Designing software components to be tested outside their normal execution environment requires enabling *seams* where inputs can be provided, and outputs can be observed. Having these seams in place enables changing the behavior of the system without changing the code. It also reduces coupling and improves modularity and reusability. *The creation of code that is testable and good unit tests does not necessarily come naturally to all software developers.* In our work with numerous companies, Construx has found that the design and unit-testing skills of technical staff generally varies widely throughout an organization.

As unit testing is deployed across an organization, Construx recommends that personnel be trained in unit-testing best practices and design for testability techniques. Some resources that can be used to help increase the staff’s knowledge of and skills in this area include the following:

- Working Effectively with Legacy Code, Michael Feathers
- Pragmatic Unit Testing in C# with NUnit, Andy Hunt and Dave Thomas
- Pragmatic Unit Testing in Java with JUnit, Andy Hunt and Dave Thomas
- Refactoring: Improving the Design of Existing Code, Martin Fowler
- xUnit Test Patterns, Gerard Meszaros
- Test-Driven Development for Embedded C, James W. Grenning
- Construx’s Developer Testing Boot Camp or Design Boot Camp



Account for unit-testing time

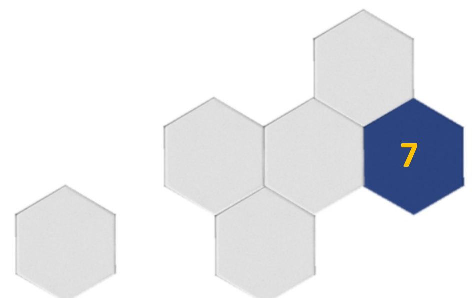
The creation of automated unit tests is not free. Management must allow for the initial time and effort to write unit tests. For developers with extensive experience with writing unit tests, these costs are usually recouped within the project due to reduced effort on rework later in the project. For most developers, though, additional time is needed to think through the process of unit testing and how to write testable software. There is a learning curve that needs to be accounted for.

In addition, developer estimates commonly underestimate the effort to write new, testable code with automated unit tests. This is especially true as unit testing is introduced to an organization.

As unit testing is adopted, it is critical that project plans and schedules explicitly and, at first, conservatively account for this risk. For teams using Scrum, the "Definition of Done" must be modified to include the creation of all needed unit tests. This will change the velocity of the team, and all stakeholders and customers need to be prepared for this change.

Collect and share coverage data

If the organization decides to collect coverage data, this information should be made available through a dashboard. It is useful to show the coverage goals, current coverage, and coverage trends (x% increase/decrease over the last month). The collection, manipulation, and display of the data should be automated.



Incrementally Improve System Testability

As unit testing is deployed, organizations typically find areas of their system(s) are difficult or impossible to unit test. These are often mission-critical areas of the system or areas that generate a high number of defects. During the unit-testing rollout, most organizations find that they need to include some level of investment in reducing technical debt and increasing the testability of the system.

Determining testability

To improve testability, it is useful to first understand the current testability of a system or component. One common approach is to assess it using the SOCK[3] model. This assesses testability in terms of

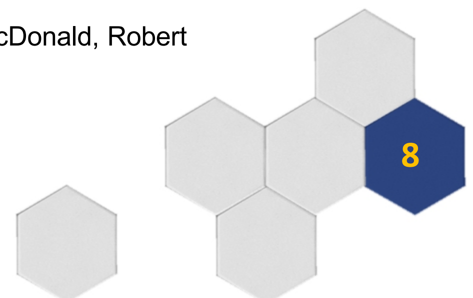
- Simplicity**.** How easy is it to derive and execute tests?
- Observability**.** How easy is it to observe the effects (outputs, post-conditions)?
- Controllability**.** How easy is it to control the inputs (pre-conditions)?
- Knowledge**.** How do you determine if a given observed behavior is correct?

Identify error-prone components

Most organizations find that adding unit testing for error-prone components is extremely valuable. These are typically the components that have a high level of field-reported (or system-test reported) defects, are difficult to expand, and/or are brittle. Identify these areas to develop a list of problematic components and ensure that time and effort is spent to fix those areas. Common ways to approach this are:

- Use Pareto Analysis to track where clusters of defects arise in a system. Such tracking usually uncovers patterns that follow the 80/20 rule: 80 to 90 percent of the defects often come from 10 to 20 percent of the code.

[3] *The Practical Guide to Defect Prevention*, Marc McDonald, Robert Musson, and Ross Smith



- Analyze the code base by looking for routines with the highest complexity counts. Numerous tools can be run on a code base to measure cyclomatic complexity, excessively long routines, deeply nested routines, etc. Running these tools will help identify routines and components that are likely difficult to maintain efficiently. These are often error-prone areas of the system.

Identifying error-prone components helps focus the creation of unit tests, as well as any necessary design and code improvements, on the most problematic areas.

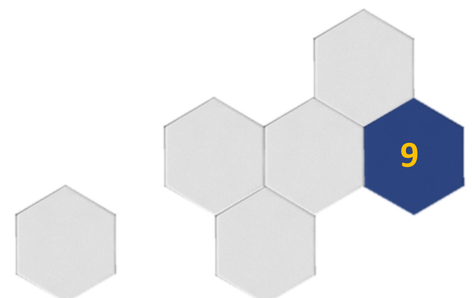
Incrementally refactor the system

Legacy systems often have areas that are difficult to automate. For system testing, a common problem is that the user interface/presentation layer is not separated from the business logic. This means that system testing must be done from the GUI, which results in test cases that are more brittle and harder to maintain. For unit testing, system components might be deeply coupled and thus hard to test in isolation.

As organizations invest in automated unit testing, they often need to incrementally refactor the system to increase the testability of system components. This refactoring is critical because testable components make it easier to create unit tests, increase the effectiveness of unit tests, and make automated smoke and system testing feasible.

As part of this process, it is important to understand the state of the current system. Legacy code can be highly coupled, and individual units cannot be tested in isolation. As the organization begins to refactor error-prone components or reduce technical debt, it needs to evaluate and understand the implications of those changes and to understand the dependencies between the different parts of the system. Work then needs to be done to determine how to separate the dependencies so that the parts of the system to be modified can be changed in isolation.

Construx recommends that the teams working on each system develop a “Technical Debt List.” This list records the changes that need to be made in each component to increase its testability. It can include changes needed to separate major components or parts of the system to enable the development of unit tests. The list should be used during project/increment planning to ensure that refactoring is included in the anticipated work.



Reduce technical debt in conjunction with unit testing

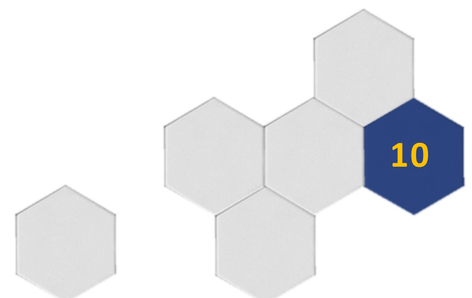
As unit-testing improvements occur, most organizations can leverage this effort to simultaneously reduce technical debt with little or no additional cost. For example, a monolithic device driver with complex and unwieldy routines is difficult both to test and to maintain. On the other hand, modular designs improve testability and maintainability. Refactoring problematic areas not only makes the software more testable, it also reduces the organization's technical debt.

Use Root Cause Analysis to identify possible improvements

Root Cause Analysis (RCA) is an especially useful technique for systematically improving individuals' abilities to design testable software. Developers typically figure out how to fix a reported defect, often discovering and fixing the root cause of the defect. However, most developers do not look into the root cause of why the defect was not discovered during the development phase in the first place.

RCA is a good technique for developers to use to gain insight into weaknesses in the software's design and implementation. Analysis of escaped defects often helps identify areas of inadequate testing caused by issues in the source code and/or design that make it difficult to write a good automated test. Developers may find, for instance, that high coupling prohibits the effective isolation of critical areas. This discovery leads to insights related to designing testable software.

Many organizations find that conducting RCA on high-severity defects can help to identify areas where refactoring would reduce the complexity of the source code and increase the testability of the software. Issues identified by RCA are candidates for the "Technical Debt List" discussed in this white paper.



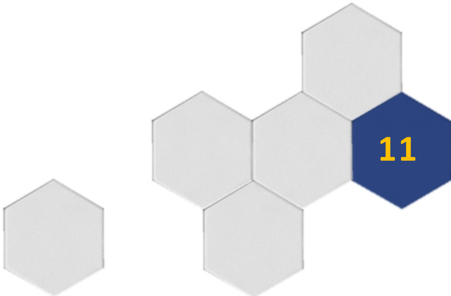
Contributors

Jenny Stuart, VP Consulting

jenny.stuart@construx.com
+1(425) 636-0108

Melvin Perez, Senior Fellow

melvin.perez@construx.com
+1(425) 636-0120



Construx

Construx Software is the market leader in software development best practices training and consulting. Construx was founded in 1996 by Steve McConnell, respected author and thought leader on software development best practices. Steve's books *Code Complete*, *Rapid Development*, and other titles are some of the most accessible books on software development with more than a million copies in print in 20 languages.

Steve's passion for advancing the art and science of software engineering is shared by Construx's team of seasoned consultants. Their depth of knowledge and expertise has helped hundreds of companies solve their software challenges by identifying and adopting practices that have been proven to produce high quality software—faster, and with greater predictability. For more information about Construx's support for software development best practices, contact us at consulting@construx.com, or call us at +1(866) 296-6300.