Improving Software Testability

George Yee, 1Z48-M

Jan 14, 2000
Contents

1. Introduction
2. Improving Testability at Design Time
3. Improving Testability at Coding Time
4. Putting it into Practice
5. Acknowledgements and References
1. Introduction

• Motivation

Poor testability --> ineffective testing --> severe penalties

- Long distance carrier loses $50,000 per minute of down time due to software failure
- Airline reservation system loses $20,000 per minute of down time due to software error
1. Introduction

• Definitions

*Testing*: selecting balls from a container
- a white ball = set of inputs giving correct outputs
- a black ball = set of inputs giving incorrect outputs

*Test effectiveness*: how many black balls are selected in n attempts - the more black balls selected, the more effective the testing

*Testability*: the likelihood of selecting a black ball in each attempt, i.e. the ease of finding a software fault

If each attempt has a high likelihood of selecting a black ball, then n attempts should produce a large number of black balls, i.e.

**High Testability --> High Test Effectiveness**
1. Introduction

• How can we improve testability?
  — Which methods can we employ in design to increase the likelihood of selecting black balls during testing?
  — We need to distinguish methods that lead to good quality code (lower number of black balls) and methods that increase testability

### Improve Quality
- modularity
- information hiding
- well-defined interfaces
- re-use
- code inspections
- testing

### Improve Testability
- focus on risky areas (location of black balls)
- increase observability (access to black balls)
- knowledge of correct values (is a ball really black?)
1. Introduction

• Software Life Cycle Focus
  — The nature of testability forces our focus on design and coding
2. Improving Testability at Design Time

- Focus On Risky Areas (Location of Black Balls)

  — Identify the risky areas of your design for special attention when writing test cases; these areas include boundary conditions (especially array out-of-bounds), memory allocation, inter-process communication, interactions with hardware, etc.

  — Identify and create test cases before writing code. Knowing what tests you have to pass before writing the code helps ensure that what you do write will address the risky side.
2. Improving Testability at Design Time

• Focus On Risky Areas - Methods

  — Create a good set of regression test cases for your components. Make it easy for these test cases to be run. Ensure that the test cases check risky areas, success paths and error paths and include the appropriate boundary checking.

  — Create CI tools that ensure the consistency of your data structures, e.g. active map vs. inactive map(s).

  — Create CI tools that ensure consistency between your data structures and hardware.
2. Improving Testability at Design Time

- Focus On Risky Areas - Methods

- Unit test / stub interactions with h/w and OS, e.g. unit test interactions with interrupt handlers using a test harness. Nothing ever happens "concurrently". One interrupt will always be processed before another.

- Plan for the use of assertions and SWERRs to observe normally inaccessible or very difficult to access critical values in risky areas (more on this later).
2. Improving Testability at Design Time

- Increase Observability (Access to Black Balls)
  
  — Always provide CI tools that allow a public software interface to be tested in isolation. For example, Connection Services (CS) has functions which are called by Traffic Protection to perform protection switching and as such CS has CI tools to independently test these functions.

  — Let CI tools have access to normally private data for reading and writing (friend type interface). Giving CI tools access to normally private data extends their reach and can potentially reveal hidden defects. Although this violates data encapsulation, it is OK for testing purposes.
2. Improving Testability at Design Time

- Increase Observability -Methods
  - Plan for lots of small functions. Each function should be at most one screen full long.
  - Plan for the use of SWERRs and assertions to observe normally inaccessible or very difficult to access values (more on this later).
2. Improving Testability at Design Time

• Knowledge of Correct Values (is a ball really black?)

  — Know what the correct output should be by studying requirements and the detailed design.

  — Know what the correct intermediate values should be by doing simulation or mathematical calculation if necessary.
3. Improving Testability at Coding Time

• Focus On Risky Areas (Location of Black Balls)

  — Test the complete set of user interfaces from some type of test head (MCEMON typically).

  — Consider the effect of multi-tasking when writing code - is there any point in time where a function will fail if another higher priority task takes over and calls this same function or some other function?

  — Macros are error prone. They are very difficult to understand and debug.
3. Improving Testability at Coding Time

- Focus On Risky Areas - Methods

  - Focus on argument checking for world visible interfaces; the degree of checking can be reduced for functions that are for internal use only. World visible interfaces are high stakes.

  - Carefully consider all flexelint information. It is a very good linter and its recommendations are typically sound (it's complaints are valid).
3. Improving Testability at Coding Time

- Increase Observability (Access to Black Balls)

  - Have separate modules for implementation and header files. This reduces compilation dependencies and promotes software layering for independent testing.

  - Use few to no globals - use functions instead.

  - Use small tight interfaces which have been agreed upon by domains very early on.
3. Improving Testability at Coding Time

• Increase Observability - Methods

— Limit the scope of automatic (stack) variables as much as possible. In “C”, create variables inside compound statements. In C++ and Java, define variables when used. This makes the variables easier to understand and thus helps to identify problematic areas.

— Put related code and data into modules with simple and restricted interfaces: encapsulate! Don't create massive files containing a mixture of unrelated code. Don't allow declarations to be in an interface if they are not required by an external customer.
3. Improving Testability at Coding Time

- Increase Observability - SWERRs
  - Use SWERRs to identify faulty areas of code.
  - SWERR with data in error path.
  - SWERR at point of failure - don't cascade them, e.g. SWERR at point of memory allocation failure - not at caller level.
3. Improving Testability at Coding Time

- Increase Observability - Assertions

  - Use assertions to reveal faults which are difficult to get at with normal testing. They are really built-in tests.

  - General Form:
    
    ```
    ASSERT (boolean expression, input to failure handler)
    ASSERT (boolean expression)   // default failure handler
    ```

  - Example:
    
    ```
    ASSERT (x>=0, x is less than zero, logfile)
    If x<0, we get warning:
    Assertion Failure. File: example.c, Line: 10, Note: x is less than zero
    ```
3. Improving Testability at Coding Time

- Increase Observability - Assertions

  — More Examples:

  ASSERT((index >= 0) && (index < max_array_size))
  ASSERT(malloc(sizeof(x)) != NULL)
  ASSERT (FALSE, expression_value, logfile)
3. Improving Testability at Coding Time

- Increase Observability - Assertions

  - The following areas are good places to use assertions:
    - in critical code
    - when indexing arrays, to avoid out-of-bounds
    - when allocating memory, to ensure space
    - at interfaces
    - prior to function calls to ensure variables passed are valid (pre-condition)
    - after function calls to ensure returning values are valid (post-condition)
    - to test that new functions are behaving correctly
    - in areas where high performance is required
    - in areas where performance values need to be known
    - in code that should not execute
4. Putting it into Practice

• A Testability Improvement Framework

Improve Testability
• knowledge of correct values (is a ball really black?)

Improve Testability
• focus on risky areas (location of black balls)
• increase observability (access to black balls)
4. Putting it into Practice

• To improve software testability:

1. Apply the methods of the Testability Improvement Framework during design and coding.

2. Implement SWERRs and assertions during coding, according to the Testability Improvement Framework.

3. Review testability improvements at design reviews (e.g. proposed CI tools).

4. Review testability improvements at code inspections (e.g. use of assertions).
5. Acknowledgements and References

• I gratefully acknowledge the contributions to the above testability improvement methods from the following GHCT design experts (in the order received):
  
  David Steele
  Andy Thomas, Tom Ross, Colin Murgatroyd
  Peter Laurich
  Paul Jean