



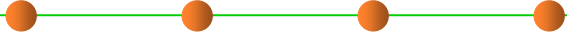
Chapter 7: Software Construction for Robustness

7.1 Robustness & Correctness

Ming Liu

April 19, 2018

Robustness & Correctness

- 
- **What are Robustness and Correctness?**
 - **How to measure Robustness and Correctness?**



1 What are Robustness & Correctness?



Robustness

- **Robustness:** “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” (IEEE Std 610.12-1990)
- **Robust programming**
 - A style of programming that focuses on **handling unexpected termination and unexpected actions**.
 - It requires code to handle these terminations and actions **gracefully** by displaying accurate and unambiguous error messages. These error messages allow the user to more **easily debug** the program.

Principles of robust programming

- **Dangerous implements** - Users should not gain access to libraries, data structures, or pointers to data structures.
 - This information should be hidden from the user so that the user doesn't accidentally modify them and introduce a bug in the code.
 - When such interfaces are correctly built, users use them without finding loopholes to modify the interface.
 - The interface should already be correctly implemented, so the user does not need to make modifications. The user therefore focuses solely on his or her own code.
- **Can't happen** - Very often, code is modified and may introduce a possibility that an "impossible" case occurs.
 - Impossible cases are therefore assumed to be highly unlikely instead.
 - The developer thinks about how to handle the case that is highly unlikely, and implements the handling accordingly.

Correctness

- **Correctness** is defined as the software's ability to perform according to its specification.
- **Robustness vs. correctness:** at opposite ends of the scale.
 - *Correctness* means never returning an inaccurate result; no result is better than an inaccurate result.
 - *Robustness* means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.
- Robustness adds built-in tolerance for common and non-critical mistakes, while correctness throws an error when it encounters anything less than perfect input.

Comparison of Robustness and Correctness

Problem	Robust approach	Correct approach
A rogue web browser that adds trailing whitespace to HTTP headers.	Strip whitespace, process request as normal.	Return HTTP 400 Bad Request error status to client.
A video file with corrupt frames.	Skip over corrupt area to next playable section.	Stop playback, raise "Corrupt video file" error.
A config file with lines commented out using the wrong character.	Internally recognize most common comment prefixes, ignore them.	Terminate on startup with "bad configuration" error.
A user who enters dates in a strange format.	Try parsing the string against a number of different date formats. Render the correct format back to the user.	Invalid date error.

"No Dashes Or Spaces"

- Credit card numbers are always printed and read aloud in groups of (usually) four digits.
- Computers are pretty good at text processing, so wasting the user's time by forcing them to retype their credit card numbers in strange formats is pure laziness on behalf of the developer.
- In Google Maps, you can enter just about *anything* in the search box and it will figure out a street address.

Pay My Bill | [Return to Bill Summary](#)

✓ Payment Type

2 Payment Details

Payment Frequency: AutoPayment

Step 2: Enter payment information

Method of Payment:

Credit Card

Name on Card:

Please enter the name on your card.

Card Number:

4111 1111 1111 1111

Please enter your card number.

Cardholder name

Cardholder name

Expiration date

Please enter numbers only.

Security code

CVC

[What's this?](#)

① Credit card number

4444 4444 4444 4

VISA

MasterCard

AMERICAN EXPRESS

DISCOVER

Billing ZIP code

ZIP

Comparison of Robustness and Correctness

- **Robustness makes life easier for users and third-party developers.**
 - By building in a bit of well thought-out flexibility, it's going to grant a second chance for users with clients that aren't quite compliant, instead of kicking them out cold.
- **Correctness, on the other hand, makes life easier for your developers.**
 - Instead of bogging down checking/fixing parameters and working around strange edge cases, they can focus on the one single model where all assumptions are guaranteed.
 - Any states outside the main success path can thus be ignored (by failing noisily) — producing code that is briefer, easier to understand, and easier to maintain.

Comparison of Robustness and Correctness

Internally, seek correctness;
Externally, seek robustness.

■ Externally and Internally:

- **External interfaces (UI, input files, configuration, API etc)** exist primarily to serve users and third parties. Make them robust, and as accommodating as possible, with the expectation that people will input garbage.
 - An application's **internal model** (i.e. domain model) should be as simple as possible, and always be in a 100% valid state. Use invariants and assertions to make safe assumptions, and just throw a big fat exception whenever you encounter anything that isn't right.
 - Protect the internal model from external interfaces with an anti-corruption layer which maps and corrects invalid input where possible, before passing it to the internal model.
- ## ■ Make your external interfaces robust, and make your internal model correct
- If you ignore users' needs, no one will want to use your software.
 - If you ignore programmers' needs, there won't *be* any software.

Comparison of Robustness and Correctness

- **Safety critical applications** tend to favor correctness to robustness.
 - It is better to return no result than to return a wrong result.
- **Consumer applications** tend to favor robustness to correctness.
 - Any result what so ever is usually better than the software shutting down.
- **Reliability**. The ability of a system to perform its required functions under stated conditions whenever required—having a long mean time between failures.
- **Reliability=Robustness + Correctness**

Terms to denote software woes

- An **error** is a wrong decision made during the development of a software system. (**error** \approx **mistake**)
- A **defect** is a property of a software system that may cause the system to depart from its intended behavior.
- A **fault**: wrong or missing function in the code. (**defect** \approx **fault**, **bug**)
- A **failure** is the manifestation of a fault during execution, is the event of a software system departing from its intended behavior during one of its executions.
- In general, **Error** \rightarrow **defect/fault/bug** \rightarrow **failure**

Error → Fault → Failure

- The **error**: write “+” to “-”
- The **fault/defect/bug**: the wrong result of statement.
- The execution will show a **failure**.


```
value1 = 5;  
value2 = 3;  
ans = value1 - value2;  
printf("The addition of 5 + 3 = %d.", ans);
```

- A programmer makes **an error (mistake)**, which results in a **defect (fault, bug)** in the software source code.
- If this defect is executed, in certain situations the system will produce wrong results, causing a **failure**.

Error → Fault → Failure

- **Not all software defects are caused by coding errors.**
 - One common source of expensive defects is **requirement gaps**, e.g., unrecognized requirements which result in errors of omission by the program designer. Requirement gaps can often be non-functional requirements.
- **Not all defects will necessarily result in failures.**
 - For example, defects in dead code will never result in failures.
- **A defect can turn into a failure when the environment is changed.**
 - Examples of these changes in environment include the software being run on a new computer hardware platform, alterations in source data, or interacting with different software.
- **A single defect may result in a wide range of failure **symptoms**.**

Steps for improving robustness and correctness

- 
- **Step 0: To program code with robustness and correctness objectives (Assertions, defensive programming, code review, formal validation, etc)**
 - **Step 1: To observe failure symptoms (Memory dump, stack traces, execution logs, testing)**
 - **Step 2: To identify potential fault (bug localization, debug)**
 - **Step 3: To fix errors (code revision)**



2 How to measure robustness and correctness?

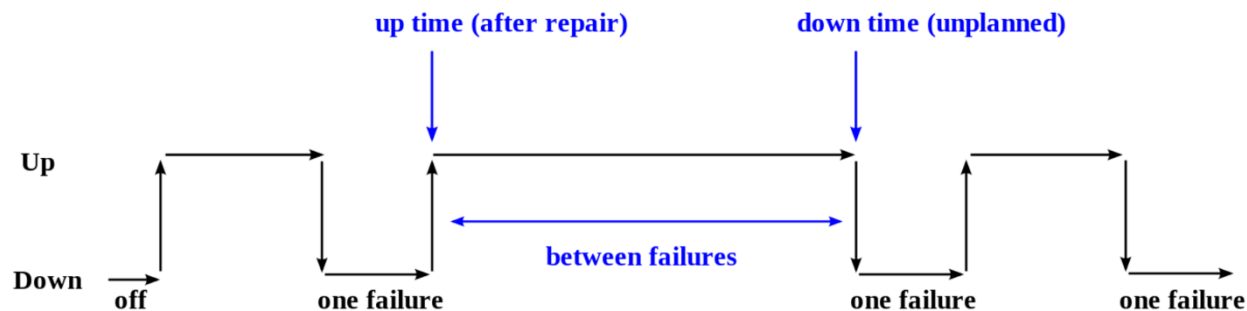


Mean time between failures (MTBF)

- **Mean time between failures (MTBF)** is the predicted elapsed time between inherent failures of a system during operation.
- MTBF is calculated as the arithmetic mean (average) time between failures of a system.
- The definition of MTBF depends on the definition of what is considered a system failure.
 - For complex, repairable systems, failures are considered to be those out of design conditions which place the system out of service and into a state for repair.
 - Failures which occur that can be left or maintained in an unrepaired condition, and do not place the system out of service, are not considered failures under this definition.

Mean time between failures (MTBF)

- **Mean time between failures (MTBF)** describes the expected time between two failures for a **repairable system**, while **mean time to failure (MTTF)** denotes the expected time to failure for a **non-repairable system**.
 - For example, three identical systems starting to function properly at time 0 are working until all of them fail. The first system failed at 100 hours, the second failed at 120 hours and the third failed at 130 hours.
 - If the systems are non-repairable, then their MTTF would be 116.667 hours.



$$\text{Time Between Failures} = \{ \text{down time} - \text{up time} \}$$

Residual defect rates

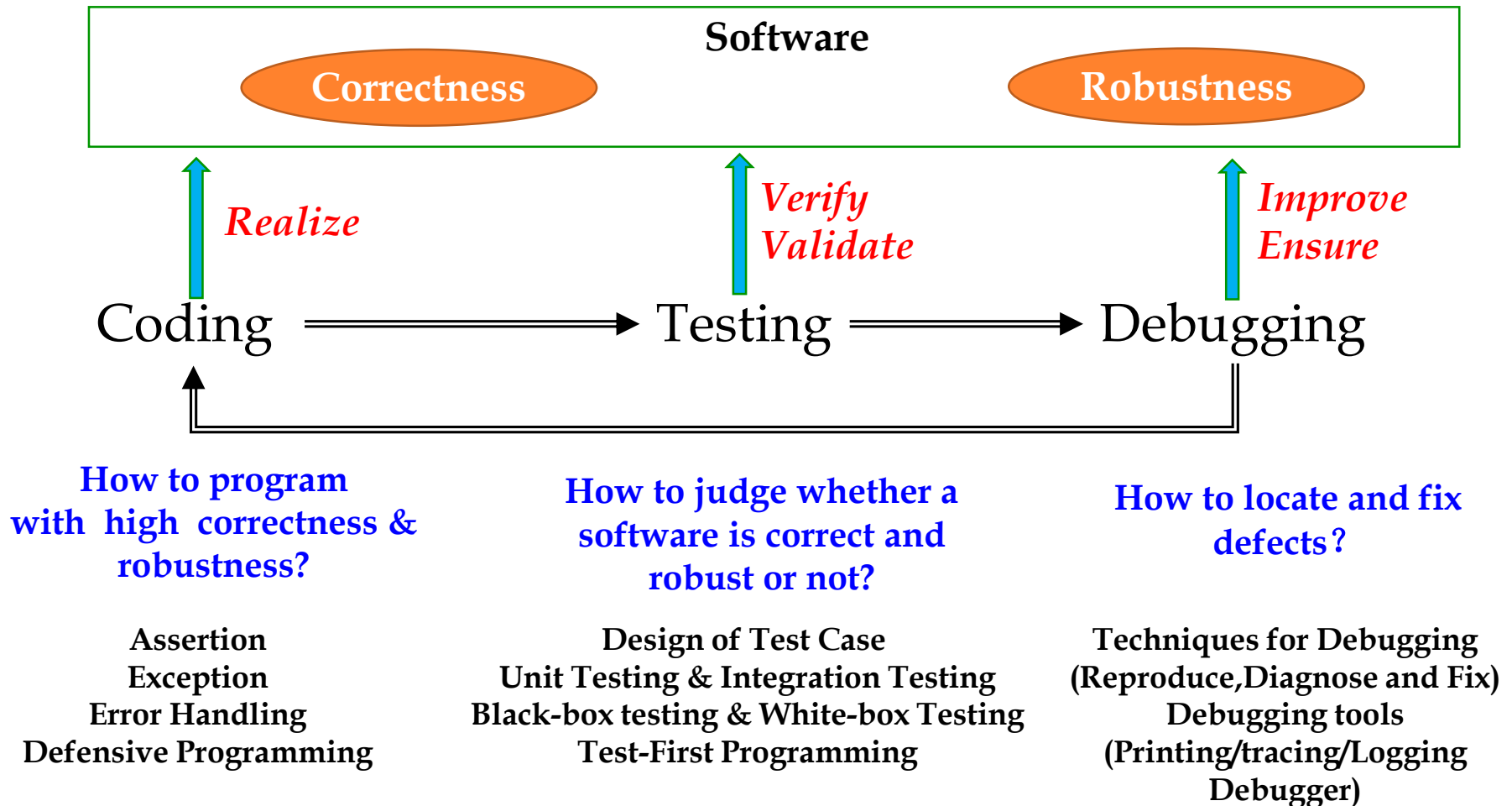
- **Residual defect rates** refers to “bugs left over after the software has shipped” per KLOC:
 - 1 - 10 defects/kloc: **Typical industry software.**
 - 0.1 - 1 defects/kloc: **High-quality validation.** The Java libraries might achieve this level of correctness.
 - 0.01 - 0.1 defects/kloc: **The very best, safety-critical validation.** NASA and companies like Praxis can achieve this level.
- **This can be discouraging for large systems.** For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!



3 Objectives of this chapter



Objectives of this chapter





The end

April 19, 2018