CS 1632 Exam 2 Study Guide

Stochastic and Property-Based Testing

**Stochastic Testing –** *Monkey Testing –* uses random processes that can be analyzed using

statistics, but not exactly tested

**Property-Based Testing –** in a stochastic testing system, you may not know exactly what the

Expected behavior should be for a given input, but you can check for properties of the system

Examples - Rather than testing specific values you want as input and output, you

can test for *Properties* you expect of your input and output.

**Property based testing**

**MORE USEFUL** for Mathematical functions, pure functions, well-specified

problems, and anything where a variety of inputs map to specific kinds of output

**LESS USEFUL** for writing to a file, network communication, text/graphic display

**Invariant –** the properties tested for in property-based testing. In the billSort method, these

Properties hold for all sorted lists. Because they don’t change, they are called *invariants*

Examples - Output array has same number of elements as input array

Every value in output array corresponds to one in input array

Value of each successive element in output is >= previous value

No element not in input array found in output array

Function is **idempotent**: same output returned for 2+ function runs

Function is pure – running it twice on same input always produces same

output array

**Falsifying the invariant -** Producing an output that does not meet the specified invariant

*Smart, Dumb, Evil, and Chaos Monkeys –*

*Dumb Monkey –* enters any old input you can think of

“SA4$@dfg52SD” “1 + 3” “#$%”

*Smart Monkey - enters inputs which a user might conceivably enter – not strictly random*

“1 + 2” “1 + + 2” “4 + 6”

*Evil Monkey –* simulates a malicious user who is actively trying to hurt your system

“ ‘) DELETE FROM entries; --“ <the entire text of war and peace>

*Chaos Monkey –* simulates what happens when things go wrong

Network topography change, system outage, someone pulls power/network cables out

**Shrinking ­**– finds the smallest possible failure, helps track down the actual issue.

**Mutation Testing** – modify the code itself in random ways

*Seeding the system under test with defects*

***Seeding*** is deliberately adding defects to a system in order to determine

whether or not out testing process is capable of catching them.

Security Testing

*Challenges in Security Testing ­*– Security testing is different from other kinds in that there is an

Intelligent adversary, also looking for defects.

**CIA Triad –** *aka InfoSec Triad –*

***Confidentiality –*** No unauthorized users read any data

***Integrity -*** No unauthorized users can write data

***Availability -*** System must be available to authorized users to read/write data

A system that meets all elements of CIA Triad under all circumstances is considered secure

**Active attack -** actually change the system under attack somehow, adding stuff, altering data

**Passive attack -** does not change system, like eavesdropping on network traffic

**Interruption –** attack on *Availability ­­*– reduces or eliminates availability of a given system

**Interception –** attack on *Confidentiality*, allows unauthorized user to read data

Keylogging, packet sniffing, looking over your shoulder as you enter data

**Modification –** attacks *Integrity* where existing data is modified

**Fabrication -**  attacks *Integrity* where additional data is added to the system

**Vulnerability -** a way in which one of the attacks can be utilized against a system

**Exploit -** a technique or mechanism which is used to compromise one of the elements

of the CIA Triad.

**Malware -** any software which is deliberately designed to have an undesired effect on a computer sytem, generally unbeknownst to the authorized user

Examples:

* + - * **Bacteria**: A program which consumes an excess amount of system resources, perhaps taking up all file descriptors or disk space.
      * **Fork bomb**: A special kind of **bacteria** which continually forks itself, causing all CPU resources to be used up creating more copies of the fork bomb.
      * **Logic bomb**: Code within a program which executes an unauthorized function, such as deleting all data on the first day of the month.
      * **Trapdoor**: A program or piece of a program which provides secret access to a system or application.
      * **Trojan Horse**: A piece of software which pretends to be another in order to trick users into installing and executing it. For example, a Trojan Horse may state that it contains different funny mouse cursors, but after installing it, it deletes everything on your hard drive.
      * **Virus**: A computer program, often small, that replicates itself with human intervention. This intervention could be something such as clicking on a link or running a program sent to you as an attachment.
      * **Worm**: A computer program, often small, that replicates itself without human intervention. For example, once installed on a machine, it may have that machine try to break into other machines and copy the code of the worm over to others.
      * **Zombie**: A computer with software installed which allows unauthorized users access to it to perform unauthorized functionality. For example, a system might have a mailer program built in which will allow other users to send spam from your machine, so that the actual senders cannot be tracked.
      * **Bot network**: A collection of zombies controlled by a master.
      * **Spyware**: Software which surreptitiously monitors the actions of the user of the system. For example, software may report back daily what all of the keystrokes of the user were.
      * **DoS Tools**: Tools which enable denial of service attacks.
      * **Ransomware**: Software which performs an unwanted action (e.g., encrypting your hard drive) and asks for money or other compensation in order to undo it. This money usually goes to the creators or users of the software, not the software itself (until artificial intelligences become more advanced).

**Common Attacks**

**Injection Attack –** attacker is attempting to get your computer to run arbitrary code.

**i.**e. SQL Injection attack

**Sanitizing –** involves ensuring that input from a user will not be directly executed, by “cleaning” it up so it can’t run.

**Examples –**eval() in JavaScript, adding “OR 1=1” to SQL, DROP TABLE, etc.

**Sanitize** by not allowing special chars in usernames, or check for them and return NO\_RESULTS\_CODE before any SQL can execute.

**Buffer Overruns –** When you must allocate a finite amount of space for data to be put

into. In Java, ArrayIndexOutOfBounds exception is thrown, but in C, no

bounds checking occurs at runtime. System keeps writing data past the

end of the array, which may overwrite executable code or other system

data. If this data is carefully crafted, it could even allow shell access to system.

**Security Misconfiguration –** Although a system may operate in bulletproof way when

set up correctly, no one is going to be as rigid as you are when setting up their version.

Example – If only way to properly set up system is to read 500 page manual and set some obscure command line switches, nearly every system you ship will be misconfigured.

**User Testing –** User testing involves having a user perform some task, often with

minimal to no guidance.

This usually is used to determine best UI for system, but can also be

done to figure out how typical users configure system, and which parts

they configure incorrectly.

**Insecure Storage –** writing sensitive data to log files, allowing users direct access to

a database, storing private keys in code, which is in a public repository

**Principle of least privilege -**  limiting users to minimal amount of

access they need to do their job.

**Social Engineering -**  involves manipulating people to underhandedly cause them to

perform actions that put the security of a system at risk.

**Phishing –** trying to get personal or other sensitive information via email or

other communication.

*The poorly constructed email is actually a screening mechanism*

**Spear Phishing -**  user is specifically targeted, using relevant details – other

users, the targets boss, perfectly forged.

**Penetration Testing –** has a user, often external to dev team, attempt to gain

unauthorized access to a system using any means at their disposal

(tempered by limits of contract, like not deleting data etc).

**General Guidelines –**

Time spent on security testing is time not spent elsewhere

If you are operating in a regulated field, ensure you are following all standards

of that field. i.e. HIPPA

Determine most important aspects of system to guard against

**Pareto Principle -**  easy to find defects are first vulnerabilities looked for by

attackers

Interacting with Stakeholders

**Stakeholder -**  any person who has a ***direct*** interest in the product

The specific kind of interest may vary based on the person and their role

* **Customers**: These are the people who will pay for the software. They will focus on receiving a

working system for a low price, and care about cost and the return on investment of purchasing

it.

* **Users**: These are the people who will use the software. Note that these are sometimes the

same as customers, but sometimes not—for example, when a school purchases class registration software, the school is the customer, but students are the users. They will care about the usability of the system and if it allows them to easily accomplish their goals.

* **Project Management**: These are the people who manage the specific project, ensuring that it

is developed at an appropriate rate and released on time and on budget. They will care about

scheduling (from a resource, time, and scope perspective) and software quality.

* **Upper Management:** Usually higher up in the management chain, to the point where they

will often not be very familiar with a project itself. They will care about the financials of the

project, such as whether or not it is likely to make a profit.

* **Developers**: The people who write the software. They will care about the tools that they

use, how well the program works from a technological standpoint, and how well it solves the

problem and meets the requirements.

* **Testers / Quality Assurance**: The people who test the software and determine the quality of

the system. They will focus on finding defects on the software and care about releasing a

quality product.

* **Support Personnel**: The people who keep the software running, such as field engineers, help

desk personnel, and deployment personnel. They will care about uptime and quality of the

system, and its ease of use on behalf of customers.

* **Assessors**: These people focus on the assessing the legality of a system, ensuring that it meets

all the demands made of it by laws for the domain in which it is operating. They will care

about meeting the legal or other documented requirements.

**Language**

**Developers -**  terms of technology

**Testers -**  terms of quality

**Project Managers –** terms of deadlines

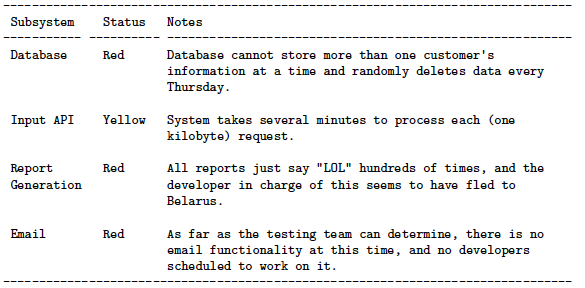
**Upper Management –** terms of financials

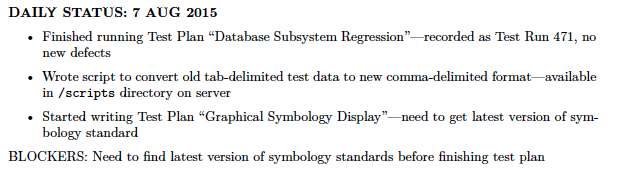
**The Red-Yellow-Green Template**

Green – No issues, on time to complete within budget

Yellow – few issues, but minor and fixable, reduced quality, needs additional resources

Red - major problem(s), issues needs substantial help to release on time, if possible



 **Status Reports**

Web Testing

Explain how web testing is done in Katalon:

Three steps:

Command – what to do

Target – To what?

Value – How? (type what?)

Common commands: **open, click, type, assert**

Assertion Types:

*assertText* - assert text present in element

*assertCookie* – assert cookie exists

*assertElementPresent* – assert element exists on page

*assertAlert* – assert an alert took place

*assertEditable* – assert an element is editable

*assertEval* – evaluate some javascript and assert the result.

How to specify a target – CSS, xpath, id, other tag

in Katalon – use *Select* tool

Static Analysis

Types:

Code Coverage – how much codebase is covered by test suite

Branch Coverage, Statement Coverage, Decision Coverage,

Parameter value Coverage, Path Coverage, Entry/Exit Coverage

Code Review / walk-through

Code metrics

Cyclomatic Complexity, Class fan-out, # lines per class, # interfaces,

number of overridden methods etc.

May help ensure no one checks in code with high cyclomatic complexity

Formal Verification

Mathematically prove behavior of program from first principles

reduce to non-turing-complete language to avoid halting problem

Slow and Inflexible

Compilers (technically)

checks syntax, uncaught exceptions, dead code,

Bug Finders

finding defects without executing code

Full of false positives

Linters

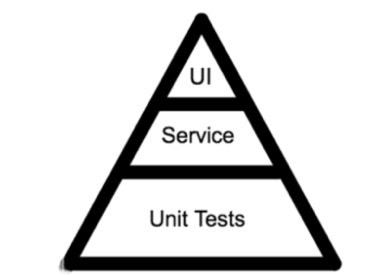
allow entire team to use consistent spacing, tabs, variable naming, etc

Very common

Mutation Testing

Testing our tests, modifies the code or method and sees if code still passes.

Testing Strategy and Process of Quality

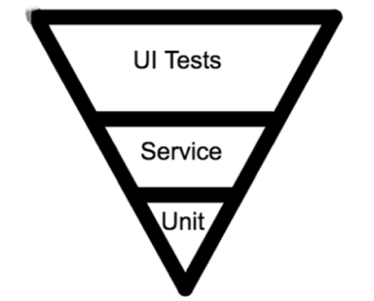
 **The Testing Pyramid**

10% UI Tests

20% Service Tests

70% Unit Tests

**Ice Cream Cone Anti-Pattern**

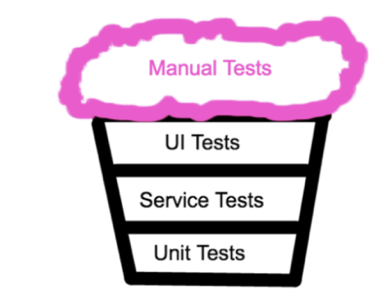


70% UI Tests

20% Service Tests

10% UI Tests

**Cupcake Anti-Pattern**



Lots of manual tests on top

**Examples of Testing Strategies:**

**Performance Testing:**

* + - Full application performance – UI Test
    - Component Testing – service test
    - Performance of a function – Unit test

**Security Testing**

* + - Penetration Testing – UI Test
    - Testing subsystems for injection – service test
    - Checking against buffer overflow in method – unit test

**Combinatorial Testing**

* + - Checking OS /Browser combinations – UI Tests
    - Combinations of Microservices Loading - Service Tests
    - Checking for results of Boolean args - Unit Tests

**Rantly**

Here is how you would express a Rantly property:

it "define your property here" do property\_of { Rantly { <GENERATOR GOES HERE> } }.check { |a\_generated\_value| <EXPECTATION GOES HERE> } end

Therefore, an example would be:

it "integer property only returns Integer type" do property\_of { integer # the generator }.check { |i| # i is the generated value expect(i).to be\_a(Integer) # the expectation } end

## A Failing Test and Shrinking

Let’s see what Rantly tells us when a property fails. We will tell Rantly to create arrays of integers, and then check that every generated array has completelyeven elements. Obviously, that will fail. The interesting thing is how will it fail? Let’s write the shady property in **spec/array\_spec.rb** :

require 'rantly'

require 'rantly/rspec\_extensions'

require 'rantly/shrinks'

RSpec.describe "Array" do

it "even numbers" do

property\_of {

Rantly { array { integer } }

}.check { |i|

expect(i).to all(be\_even)

}

end

end

When you run the file, this is the output:

[0, 0, 0, 0, -1324248444, -819907805037675589]

found a reduced failure case:

...

[0, 0, 0, 0, -10102, -819907805037675589]

found a reduced failure case:

...

[0, 0, 0, 0, -77, -819907805037675589]

found a reduced failure case:

...

[0, 0, 0, 0, -1, -819907805037675589]

found a reduced failure case:

...

minimal failed data is:

[0, 0, 0, 0, 0, -819907805037675589]

F

**Custom Generators: Generating a DNA Sequence**

In this example, we will create a DNA sequence generator. For our purposes, a DNA sequence is basically a array that contains a combination of A, T, G and C.

An example would be ["C", "G", "A", "G", "A", "T", "G"]. Our first stop is Rantly#choose, which let’s the generator pick a value from the specified choices:

choose("A", "T", "G", "C")

Next, we know that we need an array. The array generator accepts a block, which is called to generate an element of the array. This is exactly what we need:

Rantly { array { choose("A", "T", "G", "C") } }

To add some variation, we can also have the generator produce various length arrays by specifying a range:

Rantly { array(range(0, 20)) { choose("A", "T", "G", "C") } }

Try this out on a console. You would need to do a require "rantly":

> 10.times { p Rantly { array(range(1,20)) { choose("A", "T", "G", "C") } } }

["T", "A", "A", "G", "A", "A", "T", "G", "G", "T", "A", "T", "T", "T"]

["T", "A", "T", "T", "G"]

["T", "T", "C", "G", "T", "T", "C", "A"]

["T", "C", "C"]

["G", "G", "T", "C"]

["A", "A", "A", "A", "C", "G", "T", "G", "G", "T"]

["C", "A", "G"]

["T", "G", "C", "C", "A", "C", "C", "T", "G", "C", "T", "C", "G", "C"]

["G", "C", "T", "T", "T", "A", "C", "A"]

["G", "G", "G", "A", "C", "T", "G", "C"]

=> 10