Chapter 10– Testing

Software Engineering

Quality

Best way to obtain quality is to put it there in the first place.

Techniques for detecting errors:

- Testing
- Inspections and reviews
- Formal methods
- Static analysis

Quality

Two notions of quality:

Verification - conforms to its requirements and specifications, "building the system right"

Validation – meets users' requirements and specifications, "building the right system"

Vocabulary

Error – mistake made by a software engineering or programmer

Fault/defect /bug– condition that may cause a failure in the system, this is caused by an error

Failure/problem – inability of the system to perform according to its specifications, this is a result of a defect in the system

Testing

Purpose of testing:

- Find defects
- Assess quality

Testing cannot prove that a product works.
It can only find defects.

Who tests?

Three options for who does the testing:

1. Programmers
2. Testers
3. Users

What is tested?

Three main levels:

1. Unit testing
2. Functional testing
3. Integration and system testing

Component testing (between functional and integration) for large systems

Progression of testing

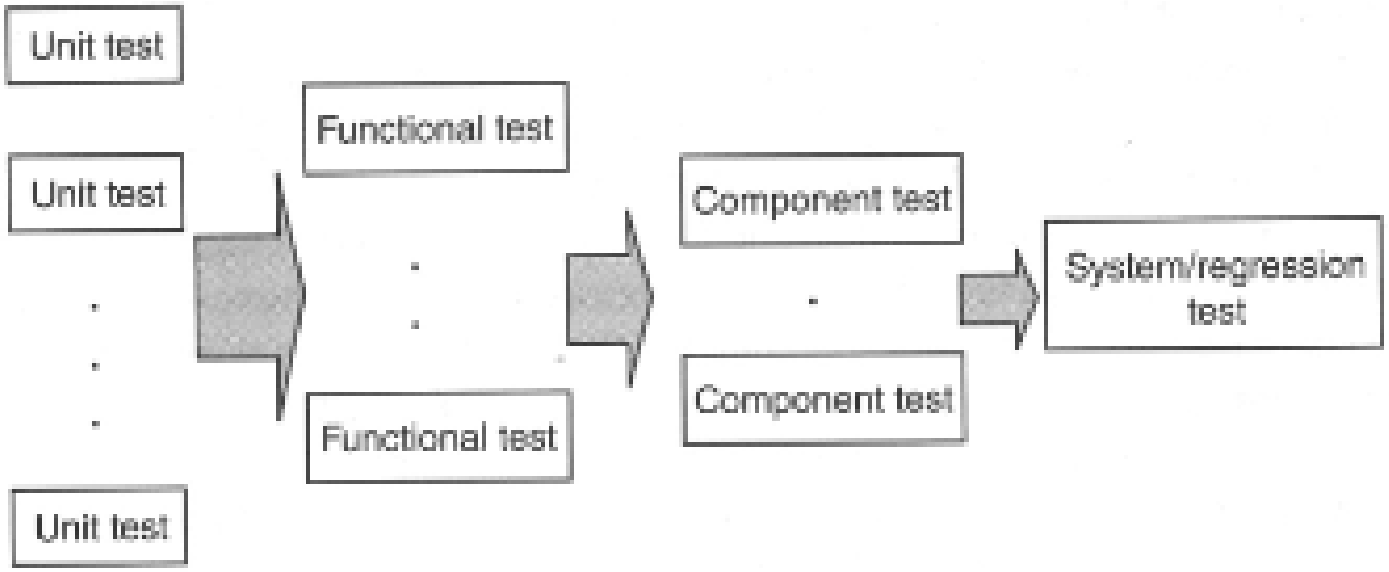
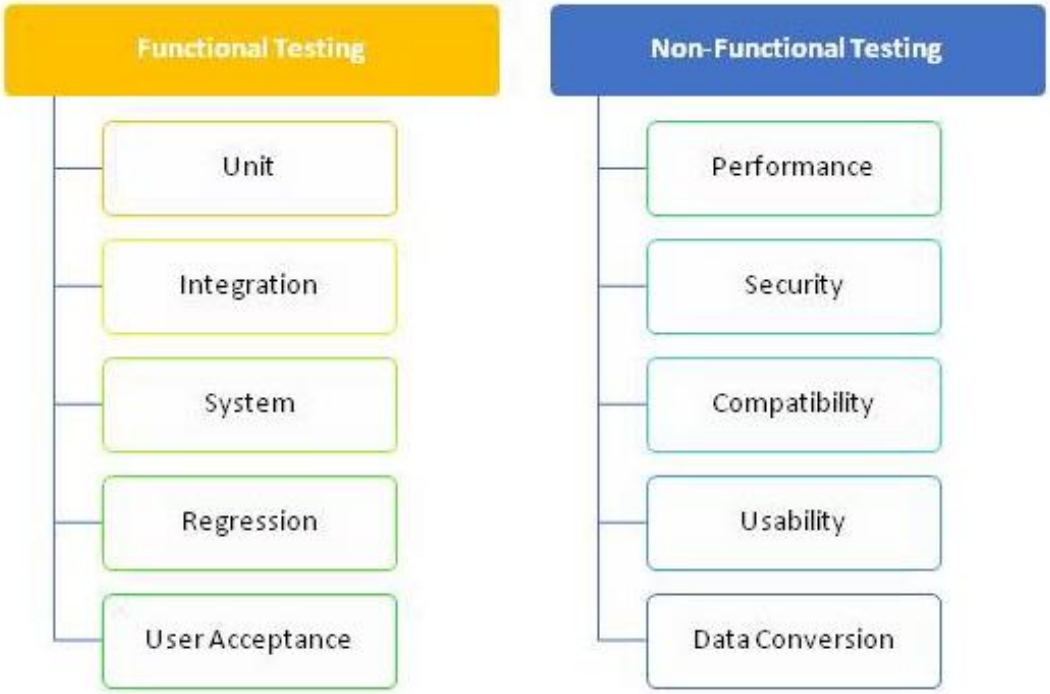


Figure 10.1 Progression of testing.

Testing – Another view



And the list goes on

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

Testing – Another view

Developers and testers test:

- Unit and integration tests - performed by developers before handing out builds to testers
- User acceptance testing – performed by testers
- Performance testing and UI testing – performed by both

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

Testing – Another view

Descriptions:

- Unit testing – smallest testable part, faking, mocking and stubbing are indispensable
- Integration – verify that components work together
- Performance testing – ensure application performs well under expected workload

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

A brief description of a few very important testing methodologies from the diagram above, that should be included in almost every test plan, are covered below.

Unit Testing involves testing individual units of source code to determine if they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. **Faking, Mocking, and Stubbing are indispensable while writing unit tests for code which has API interactions.**

Integration testing involves a combination of two or more "units" being tested. Integration tests verify that the components of the software all work together or "integrate" appropriately.

Performance testing is used to ensure that software applications will perform well under their expected workload. The features and functionality supported by a software system are not the only concerns. A software application's performance, like its **response time, reliability, resource usage,** and **scalability**, matters. The goal of performance testing is not to find bugs, but to eliminate performance bottlenecks.

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

Why are we testing?

Different types of testing for different purposes:

1. Acceptance testing
2. Conformance testing
3. Configuration testing
4. Performance testing
5. Stress testing
6. User-interface testing

How to generate and choose test cases?

Choose test cases based on:

- Intuition
- Specifications – black box testing
- Code – white box testing
- Existing test cases – regression testing
- Faults – error guessing and error-prone analysis

Vocabulary

White box testing – derive test cases by examining the code and detailed design, unit testing

Black box testing - drive tests from the requirements without consideration of the actual code

Equivalence-Class Partitioning

Equivalence-class partitioning - divide the input into several classes that are deemed equivalent for the purposes of finding errors

Black box technique

Equivalence-Class Partitioning Example

Table 10.1 Equivalence Class Example

Class	Representative
Low	-5
0–12	6
13–19	15
20–35	30
36–120	60
High	160

Boundary Value Analysis

Boundary value analysis - use same classes as equivalence partitioning but test at the boundaries rather than just a single element from the class

Black box technique

Boundary Value Analysis Example

Table 10.3 Boundary Values and Reduced Test Cases

Class	All Cases	Belonging Cases	Reduced Cases
Low	-1, 0	-1	-1
0-12	-1, 0, 1, 11, 12, 13	0, 1 11, 12	0 12
13-19	12, 13, 14 18, 19, 20	13, 14 18, 19	13 19
20-35	19, 20, 21 34, 35, 36	20, 21 34, 35	20 35
36-120	35, 36, 37 119, 120, 121	36, 37 119, 120	36 120
High	120, 121	121	121

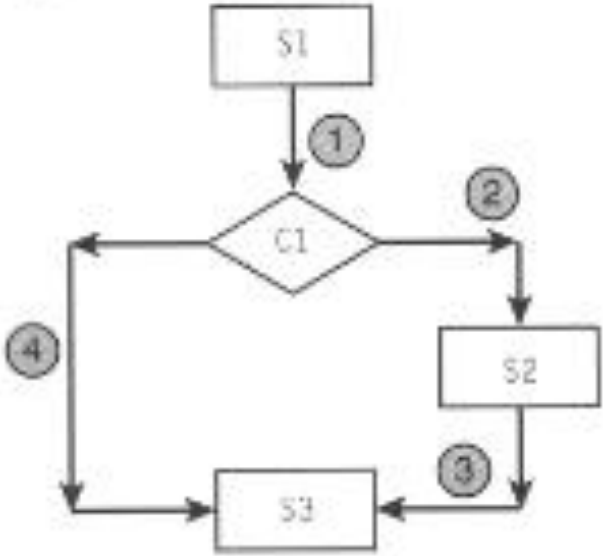
Path Analysis

Path analysis - analyze the number of paths that exist in the system or program, determine how many paths should be included in the test

White box technique

Other types:
Statement coverage
Path coverage
Decision coverage

Path Analysis Example



Path1: S1 - C1 - S3
Path2: S1 - C1 - S2 - S3
OR
Path1: Segments (1, 4)
Path2: Segments (1, 2, 3)

Figure 10.2 A simple logical structure.

Path Analysis Example 2

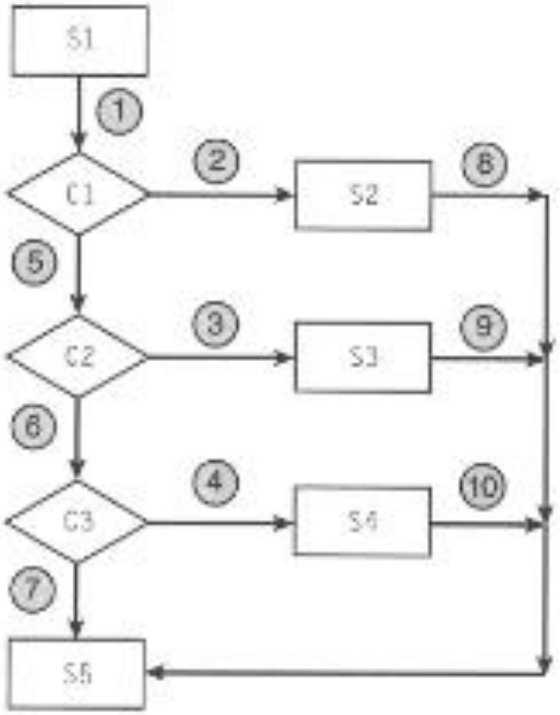
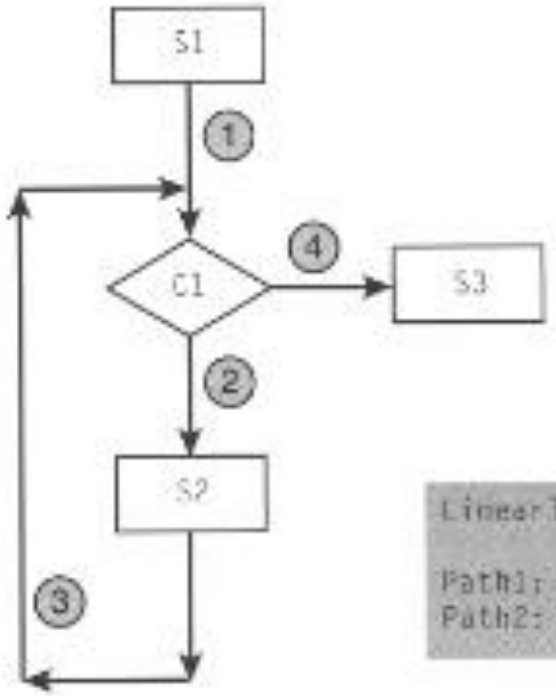


Figure 10.3 A CASE structure.

The 4 independent paths cover

Path1: includes S1 C1 S2-S5
Path2: includes S1-C1-C2-S3-S5
Path3: includes S1-C1-C2-C3-S4 S5
Path4: includes S1 C1 C2 C3 S5

Path Analysis Example 3

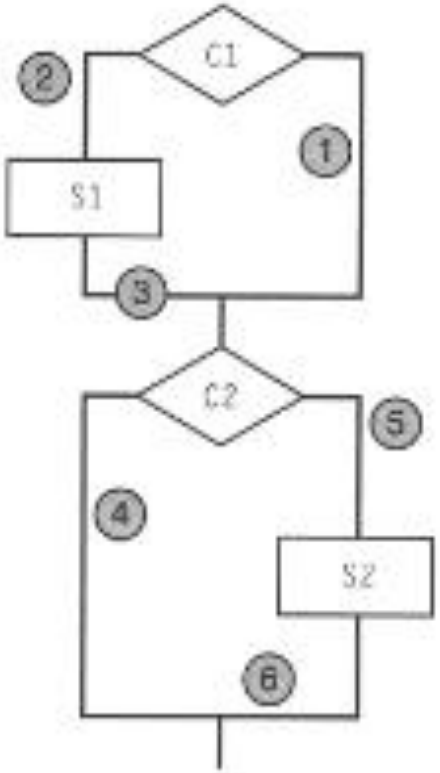



```
Linearly independent paths are

Path1: S1-C1-S3    (segments 1, 4)
Path2: S1-C1-S2-C1-S3    (segments 1, 2, 3, 4)
```

Figure 10.4 A simple loop structure.

Path Analysis Example 4



Consider Path1, Path2, and Path3 as the Linearly Independent Set

	1	2	3	4	5	6
Path1	1				1	1
Path2	1			1		
Path3		1	1	1		
Path4		1	1		1	1

Figure 10.5 A linearly independent set of paths.

Combinations

Combination testing – maybe equivalence class and path coverage

Automatic Unit Testing

Automatic unit testing using tools such as:

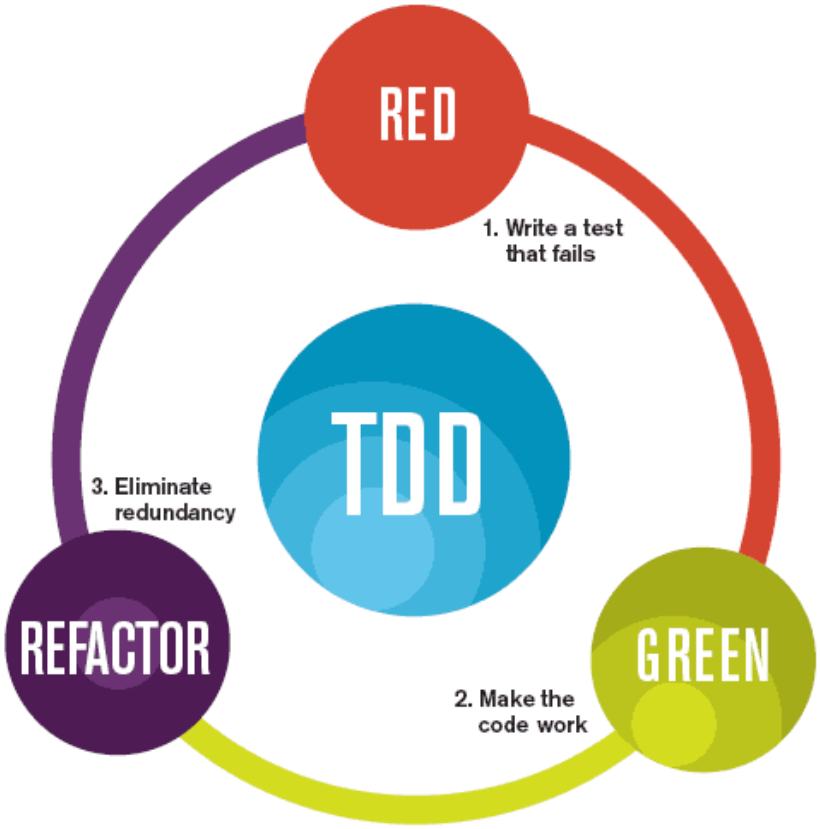
- Junit, Jtest
- PHPUnit
- Funit – FORTRAN, not F# ☺
- Nunit
- go2xunit

Test-Driven Development

Test driven development:

1. Write a test case.
2. Verify that the test case fails.
3. Modify the code so that the test case succeeds. (Write the simplest code possible.)
4. Run the test to see it works. Also run previous test cases to see nothing regressed.
5. Refactor code to make it pretty

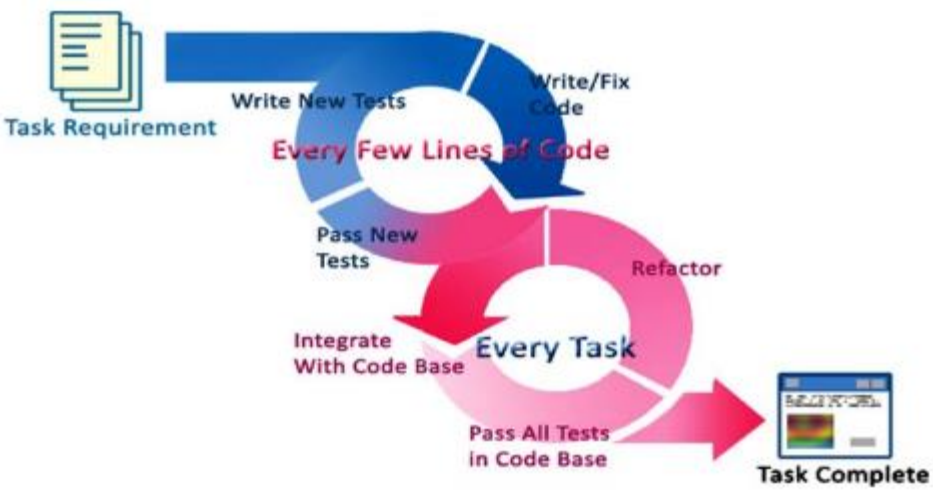
TDD



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

TDD with Integration Testing



Number of Iterations per task

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

TDD Studies

TDD case studies:

- Used TDD: 3 development teams at Microsoft & 1 at IBM
- Pre-leased defect density of 4 products decreased between 40%-90%
- 15%-35% increase in initial development time

https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/

AbOut tests – model and endpoints

user_test.go

backend > models > user_test.go > {} models > TestGetAllUsers_GoodCase

```
run package tests | run file tests
1  package models
2
3  import (
4      "testing"
5  )
6
7  var api UserRepo
8
   pin test | run test | debug test
9  func TestGetAllUsers_GoodCase(t *testing.T) {
10     // Perform the query and collect results.
11     users, err := api.GetAllUsers()
12     if err != nil {
13         t.Errorf("failed to get all users from database: %v\n", err)
14     }
15     // Some users are expected to have been returned.
16     if len(users) == 0 {
17         t.Errorf("failed to get all users from database: expected a non-zero " +
18             "value, got 0\n")
19     }
20 }
21
```

AbOut Continuous Integration

.gitlab-ci.yml

```yaml
image: dolphindalt/about-docker

services:
  - mysql:5.7

stages:
  - build
  - test

before_script:
  - mysql -hmysql -uroot -pmysql AbOut_Celia < db_docs/AbOut_DB_createTables.sql
  - mysql -hmysql -uroot -pmysql AbOut_Celia < db_docs/AbOut_DB_populate.sql
  - mysql -hmysql -uroot -pmysql AbOut_Celia < db_docs/AbOut_DB_storedProcs.sql

variables:
  MYSQL_DATABASE: AbOut_Celia
  MYSQL_ROOT_PASSWORD: mysql

cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
  - $GOPATH

build_job:
  stage: build
  tags:
  - about-ci
  script:
  - cd backend/
  - go mod download
  - cd ..
```

When to stop?

Two popular techniques to determine when to stop. In both cases determine an acceptable rate before starting

1. Track testing results and use the statistics
2. Error seeding

When to stop? Track results

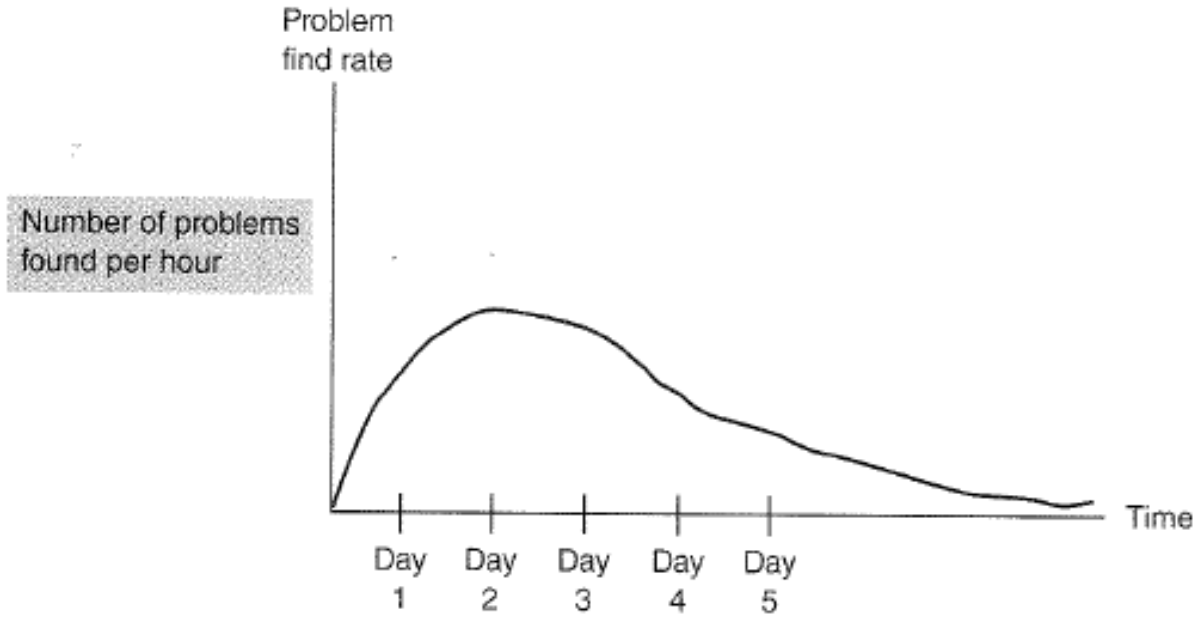


Figure 10.7 Decreasing problem find rate.

Stop when the problem find rate reaches pre-specified level, such as 1/100 hours of testing

When to stop? Error seeding

Example:

Seed 10 errors

Testing finds 52 errors, 7 seeded and 45 others

7/10 – 45/Real defects

Real defects = 64, so expect 19 more errors

When to stop? Error seeding

Example:

Seed 10 errors

Testing finds 52 errors, 7 seeded and 45 others

7/10 – 45/Real defects

Real defects = 64, so expect 19 more errors

Formal Methods

Formal methods – mathematical techniques used to prove properties of programs

Examples formal languages:

- Z
- VDM
- Larch

Formal Methods

Formal methods:

- Require considerable effort to master techniques
- Mental discipline can be extremely useful
- Applies to verification not validation
- Applies to modules that need high reliability
- Use with other techniques

Static Analysis

Static analysis – examination of the static structure of executable and non-executable files in an attempt to detect error prone conditions

Static Analysis RESTful Applications

"REST applications are designed not coded"

REST API Design Rulebook by Mark Masse, page 87