[Sample] Test Strategy

Penguinism

[note]

This document represents a sample risk analysis and test strategy that might represent a mainstream ecosystem*.*  This is just a sample intended to demonstrate typical considerations for various areas of strategy development around test concepts. As such, every situation is different and should be strategized accordingly.

# Introduction

Penguinism is a two-tiered application representing the concentration or pelmanism game where cards are turned over until matches are found. Once matches are found, the cards remain face-up and the process continues until no face-down cards remain. The application architecture is a simple two-tier architecture incorporating a stand-alone application programming interface (API) layer as well as a segregated front-end user interface layer. The self-contained API is written in C# Web API (albeit the original demo app was written in Node). The UI layer is written in Node/Express. Each layer can exist independently of the other, but both require each other for the game to fully function.

[This section would incorporate any other details of the application pertinent to the risk assessment and the strategy.]

# Risk analysis

This section outlines the overall risk factors incorporated into the entire product lifecycle. Since this application is for demo purposes, the statements made below are fictitious samples representing the characteristics of a typical end user and team.

## End user

The Penguinism application incorporates a target end user that necessitates a high degree of performance and ‘uptime’ for the system. As a result, the system is categorized as high risk in the areas of performance and availability. High profile bugs are not tolerated by the end user, making the requirements for testing substantially higher than a traditional low-use system. Further, various service level agreements are mandated around system bugs and performance failures.

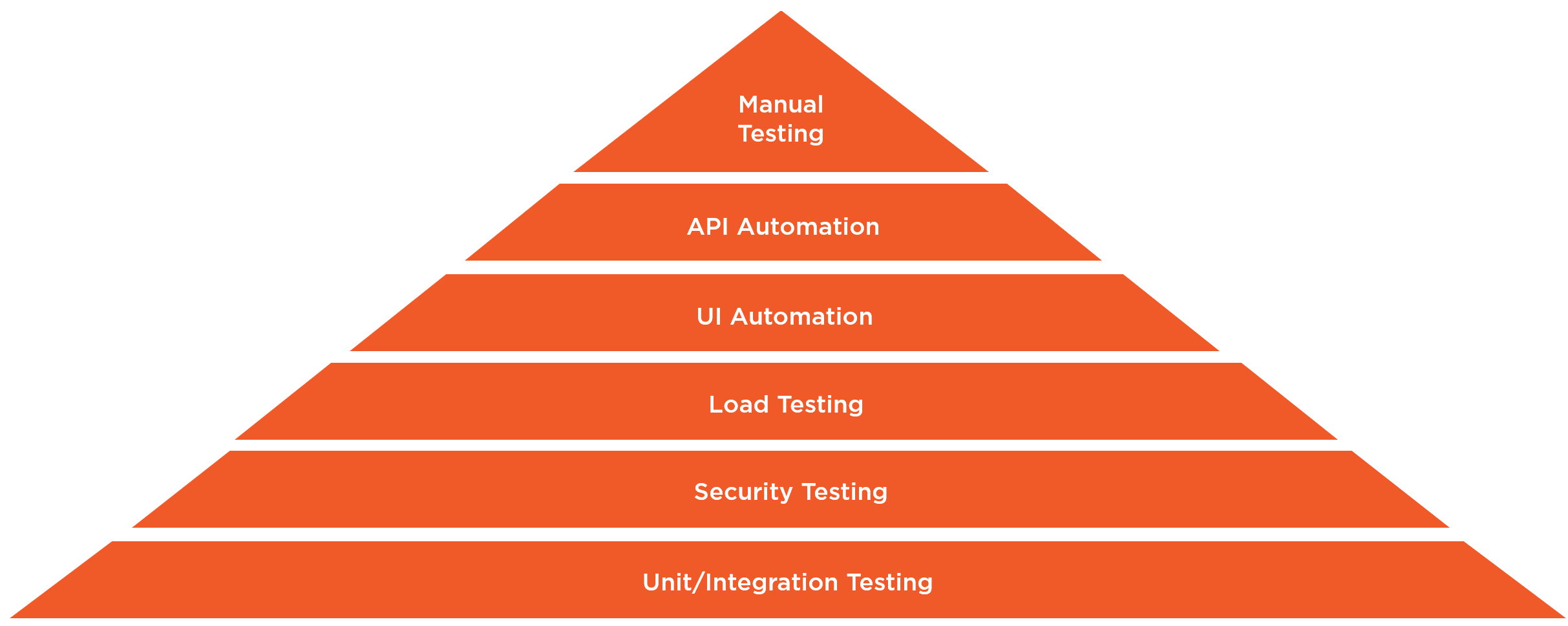
[Obviously, the fictitious section above represents that the application must be bug free, available, and generally burdens a high-performance risk, contractually so.]

## Team skill matrix

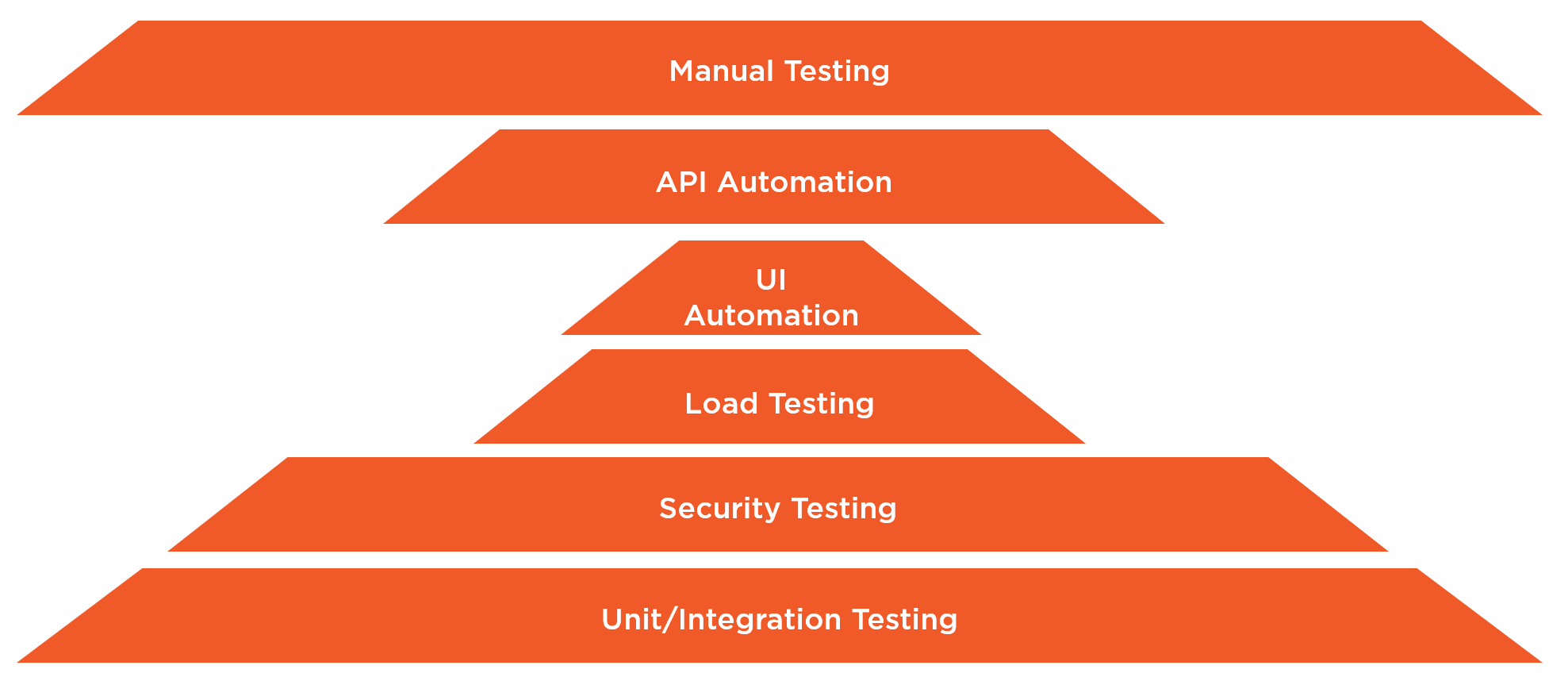
|  |  |
| --- | --- |
| **Test Concept** | **Skill Level** |
| Unit Testing | Exceptional |
| Manual Testing | Exceptional, ISTQB certified |
| API Automation | Moderate. Capable of being engineered by QA Analyst resources directly. |
| UI Automation | Moderate. Requires developer and/or QA Engineer creation. |
| Load Testing | Moderate. |
| Security Testing | Poor. Recommend using developer-level tools as well as continuous integration tools (and possibly periodic scans). |

# Test concept strategy

Due to the overall risk factors imparted by the end user, and the associated skill of the team, the following test strategy will be utilized. In short, all test concepts will be used in some manner, as illustrated by the pyramid below.



Each section below indicates the overall investment to test concepts by the Penguinism team. Although the pyramid above usually represents the quantity of each test type by the triangle size, the actual details of test quantity characteristics are outlined below. In other words, even though manual testing is the smallest triangle, it doesn’t represent the lease amount of testing in this case. A more accurate representation of testing quantity would look like this:



## Unit testing

Since the development resource skill is very high in unit testing concepts, a high degree of unit testing will be implemented. Quite simply, using code coverage tools, a 80% coverage of unit tests will be written and maintained at all times in the system.

### Developer

At the developer level, unit test tools and code coverage will be used during live development.

### Continuous Integration

Using Azure DevOps tools, builds will burden a 80% tolerance build step for all unit test code coverage. Specifically, checkins will fail builds if code coverage falls below 80%. This build step will be included in all Azure DevOps build and release pipelines, regardless of the environment.

## Manual testing

The skill level of the Penguinism team is extremely high in the area of manual testing. Due to the fact that the end user demands extreme quality measures, full suites of manual test plans will be executed and run in every environment. This will include sprint-based test plans facilitating the testing of all product backlog items, as well as critical level 1 and level 2 test cases tagged for regression throughout higher environments. As a best practice, all test plans will be executed to completion for each build. A build will not be accepted into release without all test plans passing with 100% success. Subsequent environment releases past the test environment will not be approved without 100% success of all regression test plan executions.

## API automation testing

API automation is included in the system as a result of the following factors:

* Application and end-user criticality
* API simplicity
* API targeted use by product owners in the future by many systems
* Moderate team skill in API automation engineering (QA Analysts are able to create API tests)

All factors contribute to a need to include automated API testing into the system. 100% of all API tests are included in the continuous integration build using an Azure DevOps pipeline with the SoapUI functional test task. Due in part to the fact that the API is extremely lightweight, all CI builds will deploy the sources to a development staging environment against which tests will execute. Any failure of any API tests will immediately fail the build.

Additionally, 100% of all API tests must pass in tandem with the manual regression test plans against all higher level environments before any release will be accepted into subsequent environments.

## UI automation testing

Due to the random nature of the Penguinism system, only basic UI automation tests are to be included. Penguinism has been tasked to run in all major browsers, including responsively so. As such, a smalls UI automation tests is necessary to validate the cross-browser viability of the system. Similar to API automation, all UI tests must pass against each higher level environment in tandem with regression before a regression pass is deemed successful.

## Load testing

Although the needed availability of the deployed system in production is high, the pure performance requirements out of the box far surpass the needs of the end user. However, for purposes of future capacity planning and analysis, a load test will be run prior to each release into production. The team operates on two-week sprints, with releases each third week. As such, one load test will be executed monthly. Since performance requirements are amply met out of the box, any degradations in performance will be layered back into team sprints as low priority bugs to be fixed in a future release.

## Security testing

The end user includes high requirements for security in the system. To be specific, a service level agreement has been contracted stipulating that all OWASP top 10 vulnerabilities must be met at all times. In addition, no medium or high priority issues may be deployed in code. Finally, no more than 20 low priority issues may exist at any one time.

As a result of these stipulations, three layers of security testing are included into the engineering processes.

### Developer

All developer environments use Veracode Greenlight during development. The shared settings file dictates that all high and medium issues be called out explicitly and attempted builds fail if any exist. This is due in part to a low skill level by the team on security concepts. Specifically, the team has not yet worked on a system requiring high security mandates.

### Continuous Integration

During CI builds, the code is deployed to a development staging environment against which OWASP Zed Attack Proxy scans the source. All aforementioned tolerances fail the build in the event of failure.

### Periodic Scanning

Due to the strict SLA for security, periodic manual scanning is also scheduled with each production release. This completes the three-tiered scan process in a monthly scan exercise. If any SLA-violating issues are found, an immediate urgent bug is created and incorporated into the release (or hot fix).

## User acceptance testing

As a last measure of polish, User Acceptance Testing (UAT) is performed against staging environments prior to production release. This follows an abbreviated version of the regression test plan, as well as various exploratory testing sessions by the UAT representative of the end user. This final pass through the system allows for the final ‘catch’ of any unexpected items in the system. While issues are not expected at this point, environmental and other hardware issues are possible prior to production deployment.