Test Practice Strategy Outline

California Department of Housing and Community Development

1. Overview

This document is intended to provide key concepts and ideas for organizing a modern and mature Test Practice in HCD. In its initial version, it provides material for discussing the current and desired/target state of the testing practices, and the roadmap. The document advocates adopting CI (*Continuous Integration*) approach and tooling for the development and testing practices in HCD, taking as the .

1.1. Main Sections

1. The section Key Elements of Modern Testing Practice identifies key concepts of modern testing practices applicable to software intensive systems and provides a short description of how they hang together
2. The section Towards a Roadmap for Building a Testing Practice provides an outline of the roadmap with respect to testing practices, starting with a summary of the current state and the target desired capabilities of the practice.

1.2. Change History

Table 1. Change History

Date	Contact	Description
10/26/2021	piotr.palacz@hcd.ca.gov	Minor corrections; generated .docx version.
10/26/2021	piotr.palacz@hcd.ca.gov	Sections 2 and 3 completed.
10/25/2021	piotr.palacz@hcd.ca.gov	Document started

2. Key Elements of Modern Testing Practice

This section describes the following key elements/practices of modern testing process for sizeable software systems:

- Testing many kinds of artifacts - rather than one (most often, the UI)
- Mechanizing Testing - rather than relying on manual testing
- Integrating Testing with Continuous Integration - rather than treating it as an activity separated from the development and environment promotions

- Adopting Testing Progression: starting with the simplest tests and proceeding to the most complex ones, if the simpler tests succeed, and stopping on failure - rather than waving defects and failures through.

2.1. Testing Many Kinds of Artifacts

Testing of complex software systems involves testing of many kinds of artifacts that are the building blocks for the target system. Often, each kind of artifact - such as Java class, REST API client, HTML/CSS/javascript UI - require different types of testing. Moreover, testing of some kinds of artifacts is difficult or impossible to do in isolation from other artifacts or components.

The types of tests most often encountered in mature construction of complex software systems include the following:

- Unit Test: this is the least complex form of testing, where the "unit" usually means:
    - Implementation class (in OO languages like groovy or java, or hybrid ones, like javascript)
    - Implementation function (in functional languages like javascript)
- Static Testing: "static" refers to the forms of testing that do not require executing the code. Typical examples include:
    - Testing of the source for compliance with prescribed standards and/or guidelines
    - Testing for presence of security-, performance-, or reliability-affecting patterns in the source code
- Interaction Test: tests integration between two units (sometimes more than two), where the objective is to validate is two units interact in expected ways
- Integration Test typically validates if interaction of a building block with an external system, resource, or service produces outcomes as expected. There are techniques (such as *mocking*) that allow for this type of testing even in absence of the external system or resource - which, in practice, may be the case.
- (Sub)Process Test allows for testing of defined business processes/workflows, either in part or. Note that this is different from UI testing, in which UI must be present, whereas in (sub)Process testing there may be no UI available.
- UI Test is the most used form of testing that relies on actual or simulated interactions of actual or virtual user with the User Interface. In its manual form, this is a laborious and not nearly as reliable as automatic testing.
- Configuration Test: it is not limited to the specific building blocks of the target system but rather validates expected configuration of a service or an environment. For example, if the environment doesn’t have specific components installed, or components of specific versions, then the build fails before it is even started.

2.2. Mechanizing Testing of Artifact Types

Despite the differences, most test share the same basic feature: they compare the actual outcome obtained from testing the artifact to some expected pre-defined outcome. Once we can pre-define the expected outcomes, mechanization of testing becomes feasible.

Mechanization of testing has crucial advantages over manual modes of testing:

1. The tests are unambiguous
2. Test results are reproducible and reliable
3. Testing is cheap and can be executed at any time
4. It can be integrated into larger automated processes (such as *Continuous Integration*, often abbreviated as 'CI')

Mechanization of testing has an initial cost to it, as it requires writing (usually rather simple) software to test other software. Even though the testing frameworks and tooling (that have been available for more than a decade) make the task relatively simple, mechanization of testing carries a cost. However, this initial investment is paid back again and again over thousands (if not millions) of executions of the automated tests during the lifetime of the system.

2.3. Progression in Artifact Testing

Automated tests (or any tests) should be executed in a progression starting with the tests of the simplest building blocks. This approach is based on the simple idea that, first, starting with the cheapest forms of testing and with simplest artifacts makes sense; second, that in case of any failure of simple artifacts, it doesn’t probably make sense (in terms of cost and time) to proceed to testing of more complex artifacts until the defect is fixed.

In practice, the progression can mean the following steps:

- Starting with
  - Configuration Testing
  - Static Testing
  - Unit Testing
- Proceeding to
  - Interaction and integration testing
  - (Sub)Process testing
- Ending with
  - UI Testing
  - Dedicated forms of Regression Testing

The important facet of the progression is to stop the progression when defects are encountered. For example, if the environment configuration isn’t right, it doesn’t make much sense to embark on unit testing. If any of the unit tests fail, then it doesn’t make much sense to continue with interaction testing. What does make sense is to stop before the next step, collect the defect data, and to notify the authors (or the last modifiers) of the failing artifact about the defect. The testing can resume (or restart) when the defects are fixed.

2.4. Testing As Part of Automated Continuous Integration

In a nutshell, Continuous Integration in software construction is like a Business Process: it has specific steps, transitions between steps that can be made under specific conditions, and so on.

The term *Continuous Integration* rightly emphasizes the following facets of that process:

- Integration of all the steps starting from creating/modifying source code (including configuration files, schemas, etc.) up to and including the deployment of the system into Production. These steps include the following:
    - Accepting changes to the source code from developers (usually, into a version control system)
    - Building the system on a trigger (more about it below)
    - Running all automated tests if the system builds correctly
    - Deploying the built system version to the next environment when applicable (for example, promoting from DEV to TEST, from TEST to UAT, or from UAT to Production)
    - Stopping the process in case of failures and/or defects and notifying the interested parties about them.
- Doing that *continuously* when specific triggers are present. Typical triggers include the following:
    - Schedule - such as triggering the process at the end of the working day
    - Change-related event - such as commit of modifications to the version control system
    - Deployment of the newly built system from one environment to the next

In practice, all the above triggers can be used and are used. For example, triggering a build and subsequent automated testing on every commit may seem an overkill but:

- Automated testing is cheap
- Builds can be incremental and fast
- Developers have fast feedback on health of the committed changes
- Work and testing become incremental, facilitating identification, and fixing of defects.

3. Towards a Roadmap for Building a Testing Practice

Building on the approach outlined in the previous sections, we can now start considering a Roadmap for building a testing practice. The roadmap needs to include the following elements:

- What is the current starting point?
- What is the desired target state?

- How to proceed from the current state to the target?
- What can be used to measure progress?

The subsections that follow discuss the above questions.

3.1. The Starting Point

Taking CASAS as an example, the current starting point can be characterized as follows:

- The UAT phase is the main testing phase in the system's lifecycle
  - Most tests in this phase are manual
  - Testing is limited to the end user perspective on the system
- Pre-UAT phase testing exists in a rudimentary form
  - Unit tests are present but:
    - There aren't many of unit tests
    - They are not part of the mandatory development process
    - Some are hard-coded to succeed
    - Their coverage is unknown (not measured)
  - Static code analysis tool (CodeNarc) is present but:
    - It is not used a lot
    - It is not clear what kind of actionable items it can produce in the current process
- None of the standard testing steps (unit, integration, interaction, (sub)process, user testing) are integrated within Continuous Integration (Jenkins)
  - The actual testing through Continuous Integration is in practice limited to the build succeeding or failing
  - There is no triggering of the build on committed change(s), or environment promotions, hence the CI is manual and not *continuous*

3.2. Target State and Capabilities

The target desired state is the described by adopting Continuous Integration-based approach to testing, as outlined in the section Key Elements of Modern Testing Practice. In summary, this involves ability to perform the following:

- Automated testing of many kinds of building blocks of the target system
- Following a prescribed process based on Continuous Integration approach
- Producing reliable measures of quality of the system under test

As indicated in section [Starting Point], none of the above is fully realized yet, even though the basic elements are already in place, including the following:

- Version control system

- Unit tests (in small numbers)
- Static analysis tooling (CodeNarc)
- Continuous Integration software (Jenkins)
- Specifications for several UI-based tests (performed manually)

3.3. Incremental Improvements

Practical adoption of the Roadmap requires incremental improvements. Most of these improvements can be adopted in parallel rather than sequentially. Moreover, they can use what is already in place, even if the element is not used to its full potential (for example, Jenkins as Continuous Integration software as used in CASAS).

- The first natural step is to automate the existing manual UI tests. Doing so helps reduce the effort required with testing of new system versions (including functional and regression testing is some form), increases reliability of that testing, and provides data for measuring quality changes.
    - Subsequent improvements typically involve:
        - Increasing the number and/or quality of the UI tests
        - Automated reporting of outcomes (using email, dashboards, and similar)
        - Using transient virtual users for testing and transient side effect, so that there are no persistent side-effects in the environment (or, at least, its central data store) after the testing has been completed.

The next major step is to adopt the Continuous Integration discipline and introduce incremental improvements in the following areas:

- Number and quality of Unit Tests
- Measuring coverage of the Unit Tests and flagging as defective unit tests that provide coverage below an acceptable threshold
- Introducing additional non-functional or non-UI tests, such integration and interaction tests, preferably with measurable coverage
- Making sure that detection of defects stops the testing progression and that information about the defects/failures reaches the interested parties automatically (e.g., by email)

Yet another step involves gathering metrics and measuring quality or progress towards higher quality of the system under test. This is discussed in the following subsection.

3.4. Gathering Metrics, Quality Measurements, and Reporting

Adoption of a mature testing discipline is not complete without being able to gather various metrics and quality-related measurements and being able to produce related reporting.

- Typical and natural metrics typically gathered in the testing process include failure/success counts and rates, both for the current run and historical data. However, this type of information becomes more useful if attached to metrics about the system and (versions of) its components under test. For example, a given number of defects indicates different scale of problems when small deltas are being tested, as contrasted with big modifications or additions to the system. Being able to make this type of distinctions requires linking of the information about test results as produced by the automated tests with branch/version information as available (typically) from the version control system.

- In addition to gathering metrics about how the system/application behaves under test, it is crucial to be able to measure quality of the tests themselves. Fortunately, there are approaches and tooling to accomplish that. The measure of tests' quality most often used in practice is the *Coverage* of the test in question, usually expressed as a percentage of the tested execution paths versus all available paths of execution in a component.
    - Measuring Coverage is crucial because one can easily produce many tests with 100% success rate with 0% coverage. In such a situation, the tests are worthless. To be useful, a test must have a good coverage rate, ideally 100%. This figure is not always achievable or desirable (because of the cost/effort involved), so that practical acceptable coverage values are in the range of 70-90% depending on the situation and adopted standards.
    - In case of Unit Tests, there is tooling that makes it easy to automatically determine coverage (in systems like CASAS). Measuring coverage of tests executing a workflow or a business process is more challenging but still worthy of attention.
- Reporting and aggregating test data is usually provided functionality in most Continuous Integration software. Even the already mention Jenkins, an Open-Source solution, provides sufficient reporting and data aggregation facilities. Moreover, standard capabilities of CI implementations typically include automatic generation and *delivery* of reports.

<end>