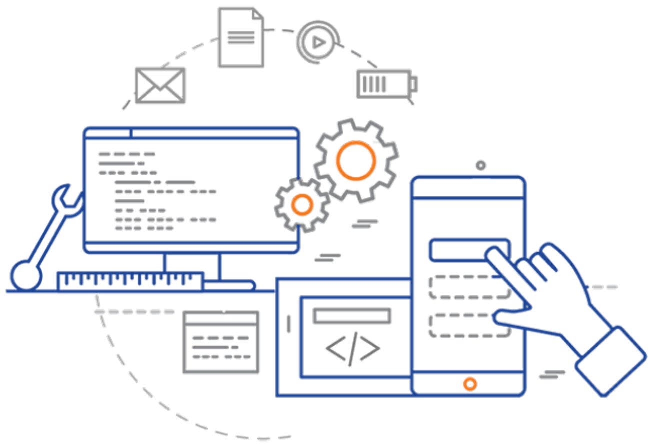


CS 398: Application Development

Integration Testing

Goals of testing; Integration tests; Dependencies

Why do we test?

The goal of testing is to ensure that the software that we produce meets our objectives when deployed into the environment in which it will be used, and when faced with real-world constraints.

Why do we test?

1. To gain confidence in the correctness of your results.

2. To gain confidence that you are handling edge cases and errors properly, which will result in a better user experience.

3. To produce an improved design, usually as a by-product of having written tests. The process of writing tests forces us to structure our code more carefully.

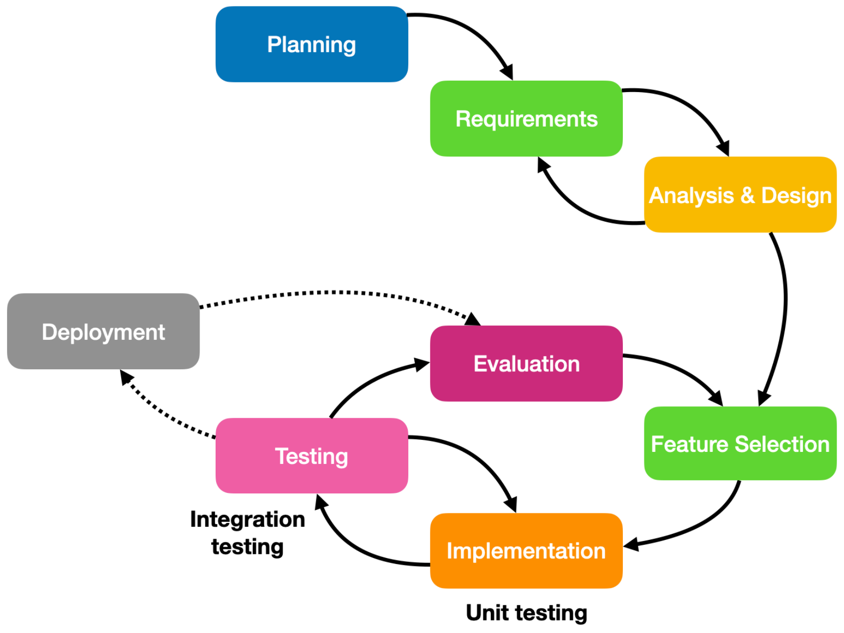
4. To help identify deficiencies and flaws in both software design and implementation.

When should we test?

Traditional views were that testing should be done after implementation. This is costly. Testing is more useful when done *earlier* in the process.

Different tests are suitable for different parts of the development process:

- **Unit Tests**: done *during implementation*, when you are working on a class.
- **Integration Tests**: done *during implementation*, when you want to ensure that classes work together.
- **System Tests**: done when features are *complete and merged*, to ensure that the system continues working.



Unit Tests

Characteristics

A unit test is a test that meets the following three requirements [Khorikov 2020]:

1. Verifies a single unit of behaviour,
2. Does it quickly, and
3. Does it in isolation from other tests.

Unit tests should target classes or components in your program. i.e. they should exercise how a particular class works. They should be small, very focused, and quick to execute and return results.

If they exercise more than a single class, they're not unit tests.

Unit test structure

Every unit test should be a separate function, consisting of the following steps:

1. **Arrange**:
   - Setup the conditions for your test.
   - Initialize variables, load data, setup any dependencies that you might need.
   - Do NOT reuse anything from a different test.
2. **Act**:
   - Execute the functionality that you want to test and capture the results.
3. **Assert**:
   - Check that the actual and expected results match.
   - Use asserts appropriately - see next page.

```
class ObserverTests() {
    lateinit var model: Model
    lateinit var view: IView

    class MockView() : IView {
        override fun update() {
        }
    }

    @Before
    fun setup() {
        model = Model()
        view = MockView()
    }

    @Test
    fun addObserver() {
        val old = model.observers.count()  // should be zero
        model.addObserver(view)
        assertEquals(old+1, model.observers.count())
    }
}
```

Public repo:
Mobile / Android / 3.MVC

Integration Tests

What are Integration Tests?

"Unit tests are great at verifying business logic, but it's not enough to check that logic in a vacuum. You have to validate how different parts of it integrate with each other and external systems: the database, the message bus, and so on." — Khorikov 2020.

A **unit test** is a test that meets these criteria:

- Verifies a single unit of failure.
- Does this in isolation from other dependencies and other tests.

Component
Class
Interface

An **integration test** is a test that fails to meet one or more of these criteria.

- It checks multiple potential units of failure.
- Tests components that have dependencies between them.
- Seeks to understand the *interaction* between components.

Components
Packages
Features

Areas of Responsibility

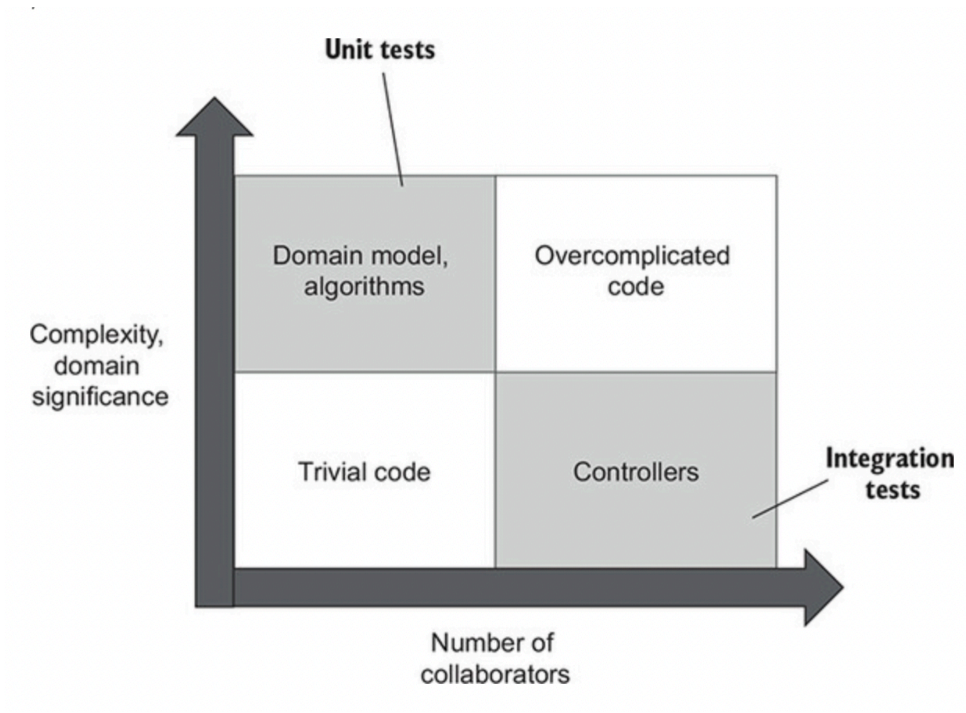
Unit tests primarily test the domain model, or business logic classes.

Integration tests focus on the point where these business logic classes interact with external systems or dependencies.

Code that we shouldn't bother testing:

- **Trivial code** is low complexity, and typically has no dependencies or external impact so it doesn't require extensive testing.

- **Overcomplicated code** likely has so many dependencies that it's nearly impossible to test.



What should I test?

Your unit tests should cover as many business scenarios as possible. You should have fewer integration tests.

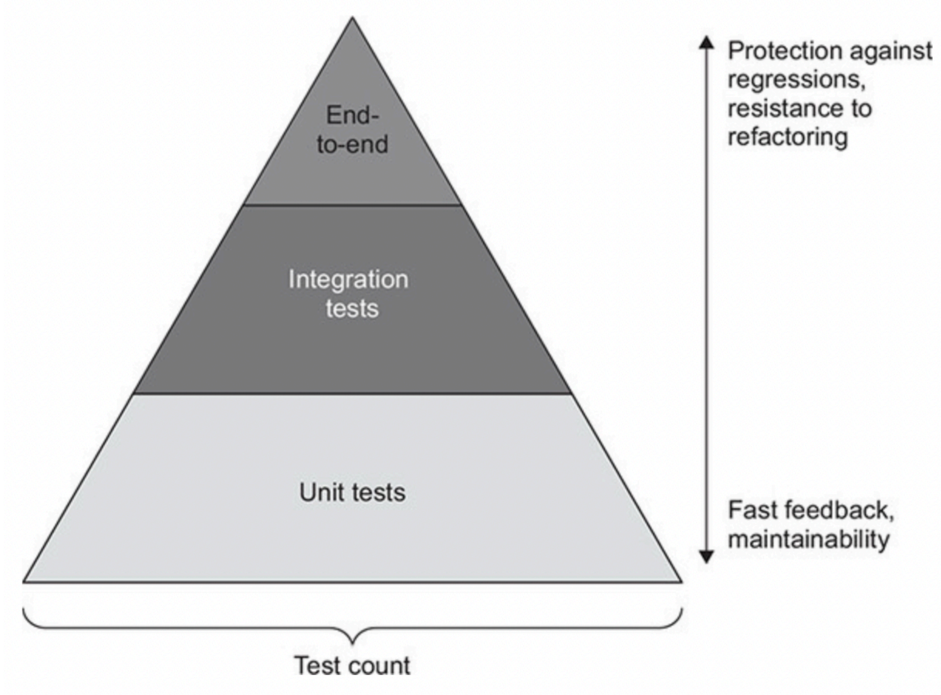
Identify a “**happy path**”: an execution path that will be your first (aka main) integration test.

- It should reflect the most commonly used execution path by your users. Ensure that common functionality works before testing more obscure functions.

- This main integration test should exercise all external dependencies i.e. libraries, interfaces, database.

The primary purpose of integration tests is to exercise dependencies.

<u>If your main integration test cannot satisfy this requirement, add more integration tests!</u>


End-to-end
Integration tests
Unit tests
Test count
Protection against regressions, resistance to refactoring
Fast feedback, maintainability

Dependencies

What is a dependency?

When you are examining a software component, we say that your component may be dependent on one or more other software entities to be able to run successfully. e.g. a library, or a different class, or a database. Each of these represents code that affects how the code being tested will execute.

We often call the external software component or class a dependency. That word describes the relationship (classes dependent on one another), and the type of component (a dependency with respect to the original class).

> *A key strategy when testing is to figure out how to control these dependencies, so that you're exercising your class independently of the influence of other components.*

Types

Managed vs. Unmanaged dependencies. There is a difference between those that we control directly (managed), vs. those that may be shared with other software (unmanaged). A managed dependency suggests that we control the state of that system.

- Examples: a library that is statically linked is managed; a dynamically linked library is not. A database could be single-file and used only for your application (managed) or shared among different applications (unmanaged).

Internal vs. External dependencies. Running in the context of our process (internal) or out-of-process (external). A library is internal, a database is typically external.

- Examples: A library, regardless of whether it is managed, is internal. A database, as a separate process is always external.

An unmanaged dependency means that we do not control its state and we may not be able to test that specific component or dependency. The best we could do it test our interface against it, since we may not be able to trust the results of any actions that we take against it.

Generally speaking, we cannot test unmanaged dependencies since we cannot control them. We also tend to be limited in our ability to test external systems, since we do not manage their state.

Test Doubles

When talking about TDD and Unit Tests, we discussed the idea of a test double or a mock.

> A **mock** is a fake object that holds the expected behaviour of a real object but without any genuine implementation. For example, we can have a mocked File System that would report a file as saved, but would not actually modify the underlying file system.

The value in a test double is to remove dependencies and allow controlled testing. As our tests get more complex, this gets more difficult to achieve.

We have 2 strategies to help reduce coupling and control these dependencies:

1. Extract interfaces

2. Introduce dependency injection

Interfaces

One important recommendation is that you introduce **interfaces** for out-of-class dependencies. For instance, you will often see code like this:

```
public interface IUserRepository
public class UserRepository : IUserRepository
```

This is common when testing, even in cases when that class may represent the *only* realization of an interface. This allows you to easily write mocks against the interface, where it's relatively easy to determine what expected behaviour should be.

Dependency Injection

Dependency injection is the practice of supplying dependencies to an object in its argument list instead of allowing the object to create them itself.

Example: Here's a class that manages the underlying database connection. How do you test the saveUserProfile() method separately from the database?

```
class Persistence {
  val repo = UserRepository() // Persistence creates the required repo instance

  fun saveUserProfile(val user: User) {
    repo.save(user)
  }
}

val persist = Persistence()
persist.saveUserProfile(user) // save using the real database
```

To reduce coupling, we could instead change our Persistence class so that we pass in the dependency. This allows us to control how it is created, and even replace the UserRepository() with a mock.

```
class Persistence(val repo: IUserRepository) {   // pass in the repo instance to decouple the classes
    fun saveUserProfile(val user: User) {
    repo.save(user)
  }
}

class MockRepo : IUserRepository {
    // body with functions that mirror how the repo would work
    // but no real implementation
}
val mock = MockRepo()
val persist = Persistance(mock)
persist.saveUserProfile(user) // save using the mock database
```

Writing Integration Tests

Start with your source code.

How do I test?

- Integration tests are identical to Unit Tests…. They just have a larger scope.
  - i.e. you are testing more than a single class, you are focused on identifying functionality that will likely span multiple classes.
- Continue using JUnit - it’s not just for unit testing.
  - See the TDD slides for setup details.

Tip #1: Reduce abstraction

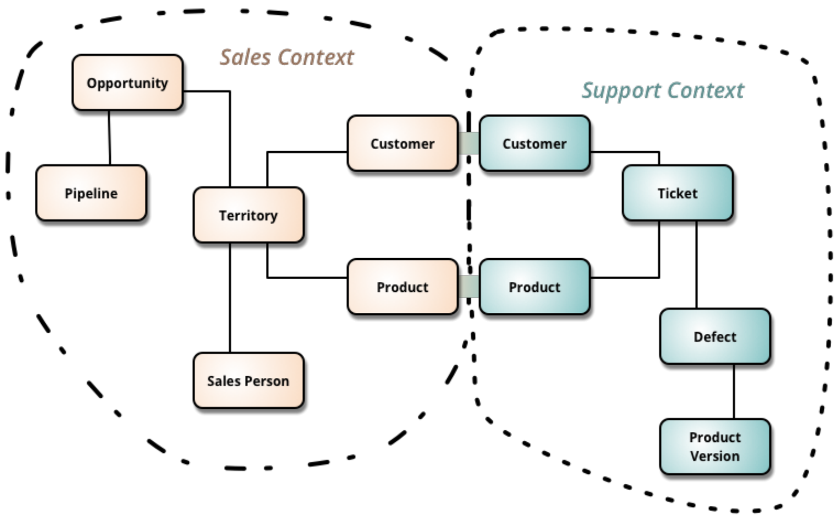
"All problems in computer science can be solved by another layer of indirection, except for the problem of too many layers of indirection." -- David J. Wheeler

- Abstraction is costly!
- A large number of abstraction layers will make it very difficult to write integration tests for your code.
    - e.g. Java libraries and "class bloat".
- Try and keep the overall number of architectural layers to a relatively small number.
    - UI, controller, persistence.
    - Domain model, application services, infrastructure.

Effective testing requires structuring your code so that it's easily testable!

Tip #2: Make domain boundaries explicit

- Think about the business logic, and the domain objects that represent your problem.
  - You want to keep your domain objects grouped in a way that makes sense.
  - Be consistent in how these objects communicate and how they relate to one another.
  - Have clean communication boundaries.
- This is no different than keeping your model and view code separate.
- It's really about being able to cleanly separate your classes for testing.



https://martinfowler.com/bliki/BoundedContext.html