Welcome

Python – TDD & Cybersecurity

DevelopIntelligence
A PLURALSIGHT COMPANY

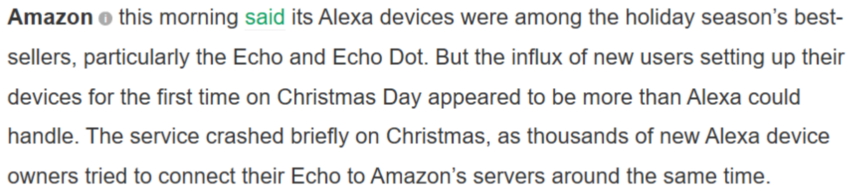
Test Driven Development (TDD)

Theory & Common Vocabulary

What Do Users Expect?

A True Story…

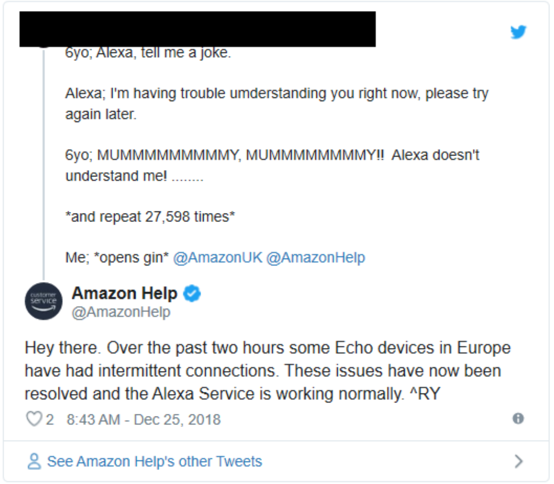
https://techcrunch.com/2018/12/26/alexa-crashed-on-christmas-day/



Amazon this morning said its Alexa devices were among the holiday season's best-sellers, particularly the Echo and Echo Dot. But the influx of new users setting up their devices for the first time on Christmas Day appeared to be more than Alexa could handle. The service crashed briefly on Christmas, as thousands of new Alexa device owners tried to connect their Echo to Amazon's servers around the same time.

Today in technology, we expect:

- Systems to be available
- Systems to be responsive
- Systems to be secure
- Systems to WORK!



[redacted]

6yo; Alexa, tell me a joke.

Alexa; I'm having trouble umderstanding you right now, please try again later.

6yo; MUMMMMMMMMMY, MUMMMMMMMMY!! Alexa doesn't understand me! ....

and repeat 27,598 times

Me; *opens gin* @AmazonUK @AmazonHelp

Amazon Help
@AmazonHelp

Hey there. Over the past two hours some Echo devices in Europe have had intermittent connections. These issues have now been resolved and the Alexa Service is working normally. ^RY

2 8:43 AM - Dec 25, 2018

See Amazon Help's other Tweets

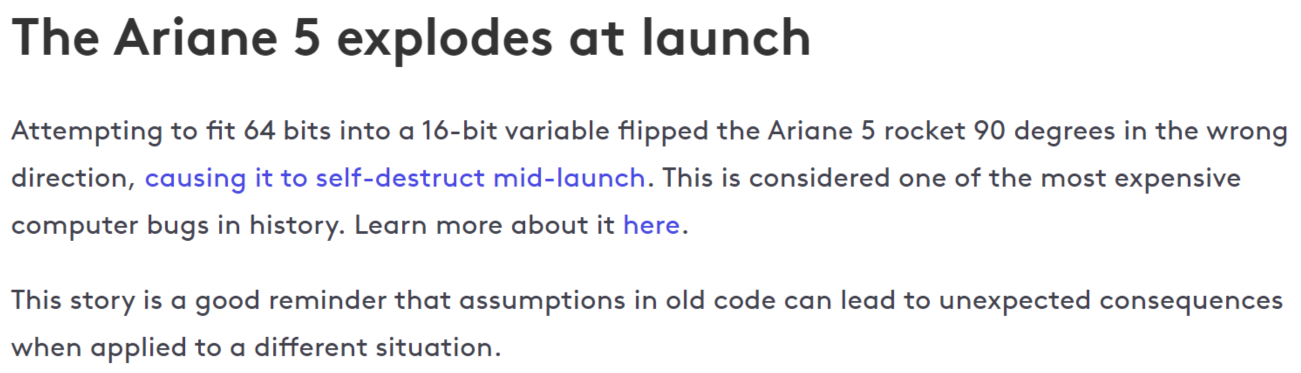
What Could Go Wrong?

Now imagine if the system in question was not a television streaming service or a website where we go for directions or a piece of AI that we can talk to and have play music on command, but instead was:

- Medical equipment in the ICU of a hospital
- The system used for air traffic control at an airport
- The system used to operate and monitor equipment at a nuclear power plant
- The system that manages all order and customer data feeding the company’s bottom-line and reporting

The criticality of system quality can increase exponentially…

What Could Go Wrong Indeed?



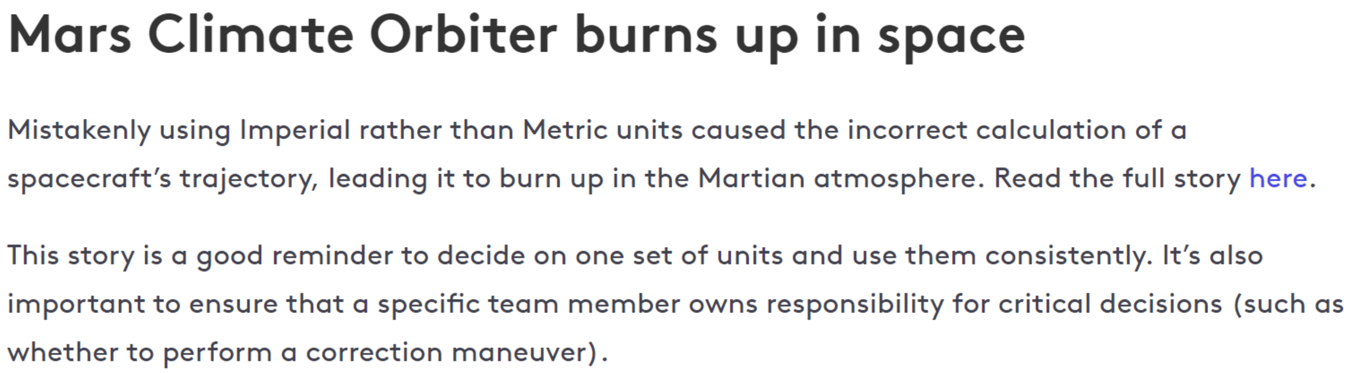
The Ariane 5 explodes at launch

Attempting to fit 64 bits into a 16-bit variable flipped the Ariane 5 rocket 90 degrees in the wrong direction, causing it to self-destruct mid-launch. This is considered one of the most expensive computer bugs in history. Learn more about it here.

This story is a good reminder that assumptions in old code can lead to unexpected consequences when applied to a different situation.

https://www.bugsnag.com/blog/4-worst-computer-bugs-in-history

What Could Go Wrong Indeed?



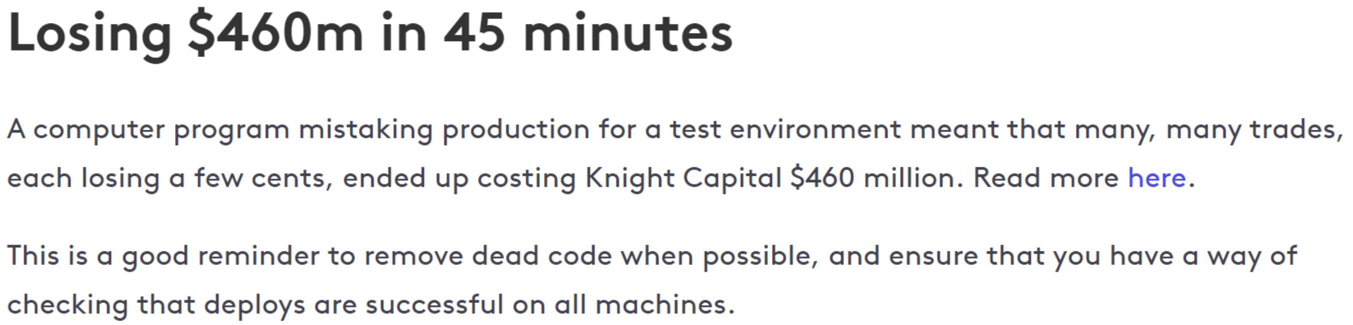
Mars Climate Orbiter burns up in space

Mistakenly using Imperial rather than Metric units caused the incorrect calculation of a spacecraft's trajectory, leading it to burn up in the Martian atmosphere. Read the full story here.

This story is a good reminder to decide on one set of units and use them consistently. It's also important to ensure that a specific team member owns responsibility for critical decisions (such as whether to perform a correction maneuver).

https://www.bugsnag.com/blog/4-worst-computer-bugs-in-history

What Could Go Wrong Indeed?



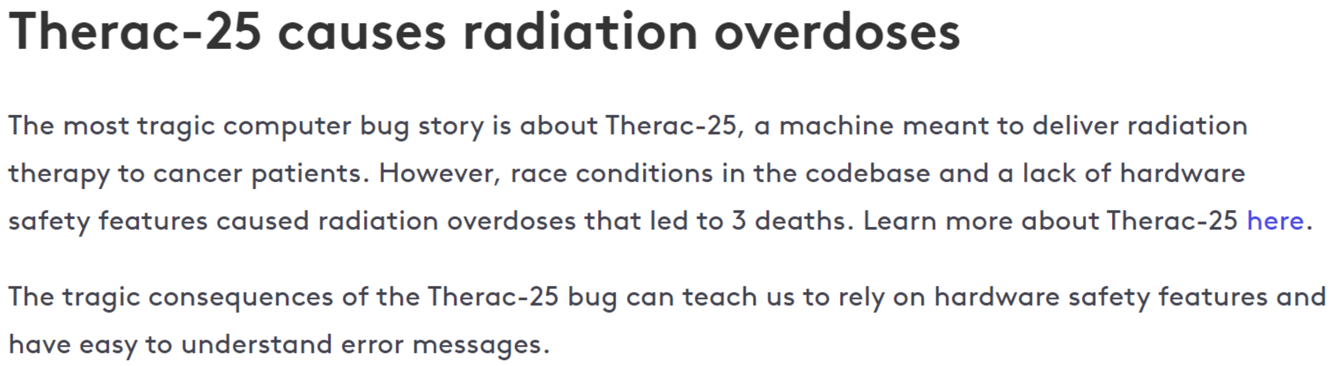
Losing $460m in 45 minutes

A computer program mistaking production for a test environment meant that many, many trades, each losing a few cents, ended up costing Knight Capital $460 million. Read more here.

This is a good reminder to remove dead code when possible, and ensure that you have a way of checking that deploys are successful on all machines.

https://www.bugsnag.com/blog/4-worst-computer-bugs-in-history

What Could Go Wrong Indeed?



Therac-25 causes radiation overdoses

The most tragic computer bug story is about Therac-25, a machine meant to deliver radiation therapy to cancer patients. However, race conditions in the codebase and a lack of hardware safety features caused radiation overdoses that led to 3 deaths. Learn more about Therac-25 here.

The tragic consequences of the Therac-25 bug can teach us to rely on hardware safety features and have easy to understand error messages.

https://www.bugsnag.com/blog/4-worst-computer-bugs-in-history

The 3 Laws of TDD (a la “Uncle Bob” Martin)

The First Law

“You are not allowed to write any production code until you have written a unit test that fails due to its absence”

The 3 Laws of TDD (a la “Uncle Bob” Martin)

The Second Law

“You are not allowed to write more of a unit test than is sufficient to fail, and failing to compile is failing”

The 3 Laws of TDD (a la “Uncle Bob” Martin)

The Third Law

“You are not allowed to write more production code than is sufficient to cause the currently failing test to pass”

Testing Approaches

Two sides of the coin (aka “Extremes”)

- Follow TDD in its “purest” form
- Don’t test (or only manual testing)

Our focus will be somewhere in-between

- <u>Unit</u> testing
- Automated
- Clean code (and clean tests)
- Red, Green, Refactor
- Testing should lead to readable/more maintainable code

Types of Testing

- Unit testing
- Integration testing
- Regression testing
- System testing
- Stress/Performance testing

Types of Testing

- $\boxed{\text{Unit testing}}$
- Integration testing
- Regression testing
- System testing
- Stress/Performance testing

F.I.R.S.T.

Clean tests are…

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-Validating
- **T**imely

Benefits

- Tests can be tied to requirements (shared success)
- Cleaner code
- Tests = system documentation
- Improved quality
- Reduced FEAR
- Regression protection
- Systematic
  - ➢ Small bites
  - ➢ Avoid premature optimization
  - ➢ Structured flow

Challenges

What are some of the challenges you face with software testing in your jobs (i.e., why don’t we test as much as we want to or should)?

Challenges

- Time pressures
- Complexity
- Requires more code
- More debugging (up front)
- More cost (up front)
- “Can’t we just ensure quality with other types of testing?”

SOLID Principles & Good Design

Architecting for the Future

- When we architect and build an application at a “point in time”, we hope that the application will continue to be utilized to provide the value for which it was originally built

- Since the “only constant is change” (a quote attributed to Heraclitus of Ephesus), we have to expect that the environment in which our application “lives and works” will be dynamic

- Change can come in the form of business change (change to business process), the need to accommodate innovation and ongoing advancement in technology

- Often, the speed at which we can respond to these changes is the difference between success and failure

Architecting for the Future

In order to ensure that we can respond to change “at the speed of business”, we need to build our systems according to best practices and good design principles:

- Business-aligned design
- Separation of concerns
- Loose coupling
- Designing for testability

Business-Aligned Design

- Build systems that use models and constructs that mirror the business entities and processes that the system is intended to serve
- Drive the design of the system and the language used to describe the system based around the business process not the technology
- AKA Domain Driven Design
- Results in a system built out of the coordination and interaction of key elements of the business process – helps to ensure that the system correlates to business value
- Also helps business and technology stakeholders keep the business problem at the forefront

Separation of Concerns

- Break a large, complex problem up into smaller pieces
- Drive out overlap between those pieces (modules) to keep them focused on a specific part of the business problem and minimize the repeat of logic
- Logic that is repeated, and that might change, will have to be changed in multiple places (error prone)
- Promotes high cohesion and low coupling (which we will talk about in a minute)
- Solving the problem becomes an exercise in “wiring up” the modules for end-to-end functionality and leaves you with a set of potentially reusable libraries

Loose Coupling

- Coupling between components or modules in a system causes problems, especially for maintaining the system over time

- System components that are tightly coupled, are more difficult to change (or enhance) – changes to one part of the system may break one or more other areas

- With coupling, you now must manage the connected components as a unit instead of having the option to manage the components in different ways (e.g., production scalability)

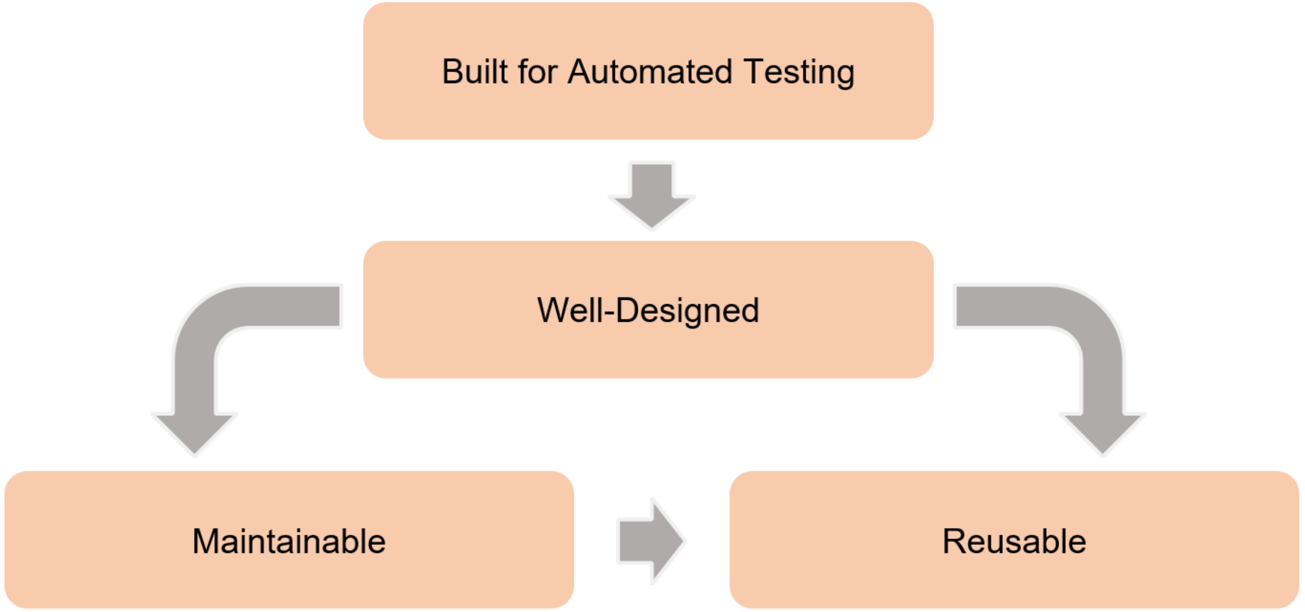
- Makes the job of unit testing the components more difficult because tests must now account for a broader set of logic and dependencies (e.g., tight coupling to a database makes it difficult to mock)

Designing for Testability

- Practicing the previous principles helps lead to a system that is testable
- Testability is important because it is a key enabler for verifying the quality of the system – at multiple stages along the Software Development Lifecycle (SDLC)
- When building a system, quality issues become more expensive to correct the later they are discovered in the development lifecycle – good architecture practices help you test early and often
- Ideally, testing at each stage will be automated as much as possible in support of quickly running the tests as and when needed

Designing for Testability

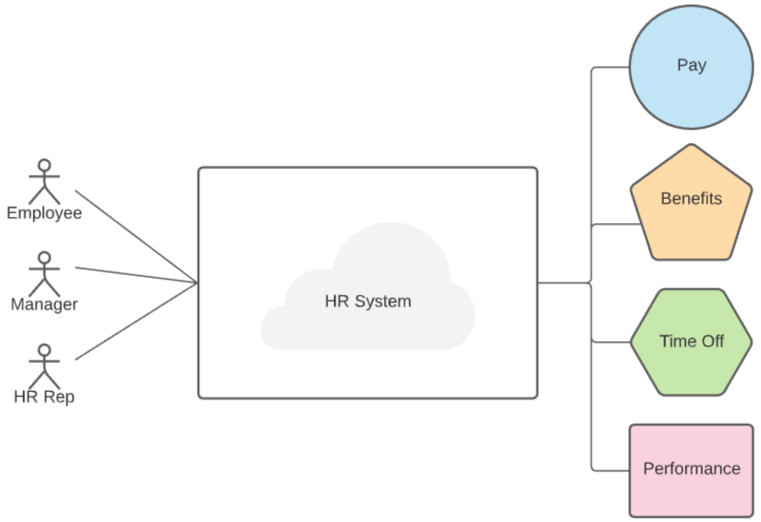
Testable code is…



Exercise – Designing for Testability

Exercise 1 (10 min) – What kind of technology or code-level tests would you expect to see with this system?

Exercise 2 (10 min) – What kind of business or end user-level tests would you expect to see with this system?



SOLID Principles

SOLID principles help us build testable code

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Glossary

Before delving deeper into each principle, let’s define some terms

- Client – A class, component or system that leverages another, separately-defined library or component for design simplification and reuse

- Encapsulation – Object-oriented design tenet that dictates that a class should hide & protect its state, providing a controlled interface for interacting with or changing that state (if required)

- Inheritance – Object-oriented design tenet that enables the building of “is a” hierarchies of related functionality through logical grouping and class reuse (including the ability to extend that functionality)

Glossary

Before delving deeper into each principle, let’s define some terms

- Polymorphism – Means “many forms”; object-oriented design tenet that uses context to determine functionality dynamically at runtime, accounting for the type of “thing” against which the functionality is being exercised

- Refactoring – Improving the internal structure of existing code without changing its external behavior or operation

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

A system module or component should have only one reason to change

Single Responsibility Principle (SRP)

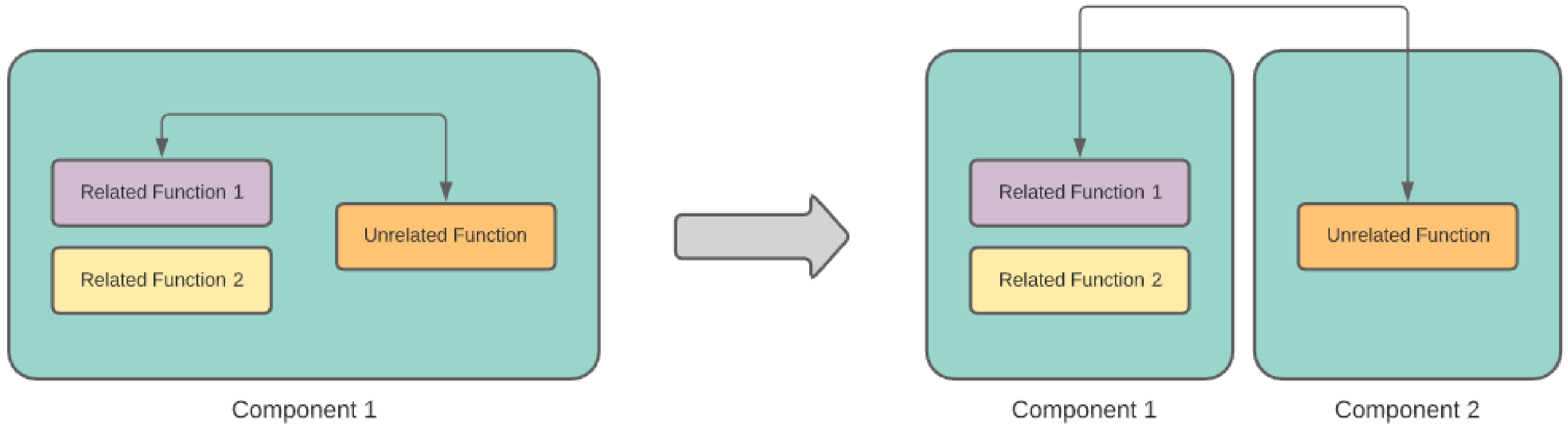
SINGLE RESPONSIBILITY PRINCIPLE
Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

Single Responsibility Principle (SRP)

Why important?

- Related functions organized together breed understanding (logical groupings) – think *cohesion*
- Multiple, unrelated functionalities slammed together breed coupling
- Reduced complexity (cleaner, more organized code)
- Promotes smaller code modules
- Reduced regression – tension of change

Single Responsibility Principle (SRP)



Single Responsibility Principle (SRP)

How to practice?

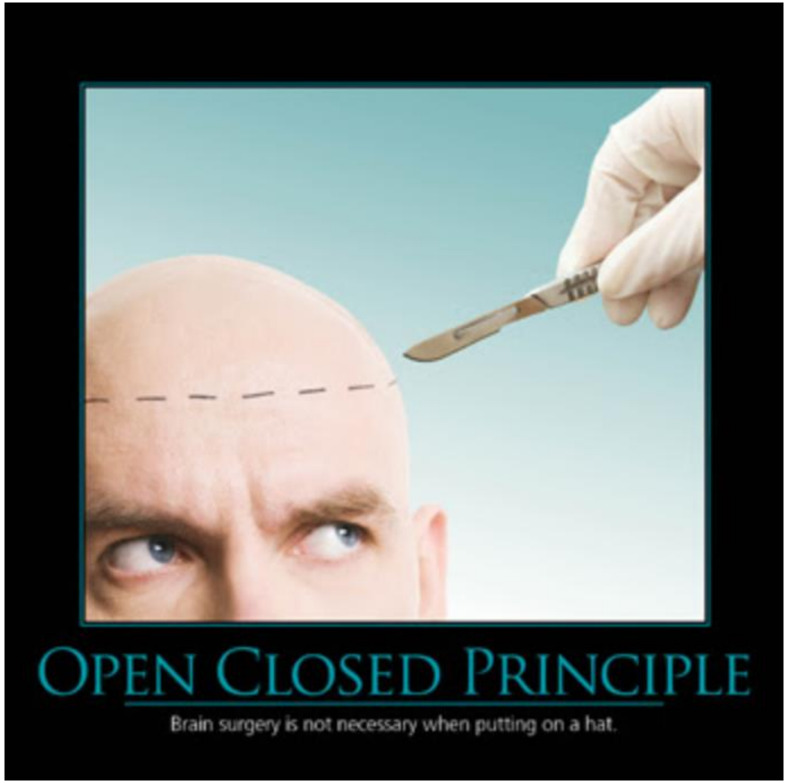
- When building new modules (or refactoring existing), think in terms of logical groupings
- Look for “axes of change” as points of separation
- Build new modules to take on new entity or service definitions – provides abstraction
- Structure clean integrations between separated modules
- Use existing tests (or build new ones) to verify a successful separation

Open-Closed Principle (OCP)

Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

Open-Closed Principle (OCP)



Open-Closed Principle (OCP)

Why important?

- Our systems need to be able to evolve
- We need to be able to minimize the impact of that evolution

Open-Closed Principle (OCP)

How to practice?

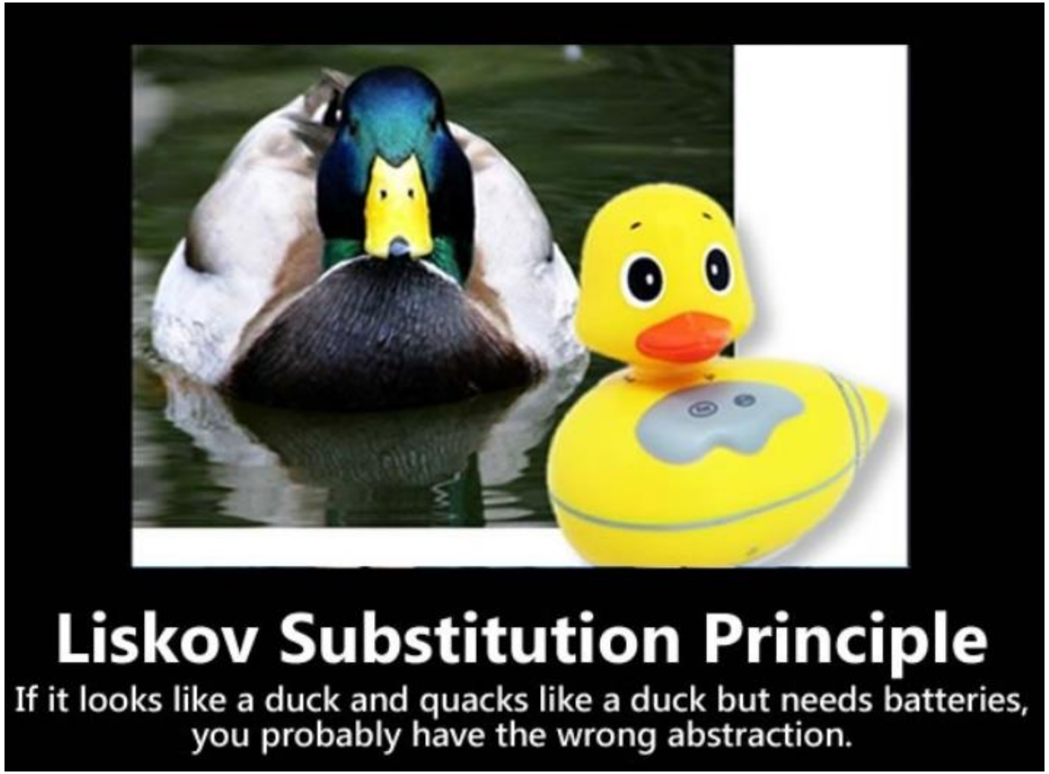
- When building new modules (or refactoring existing), leverage abstractions
- Use the abstractions as levers of extension
- Use existing tests (or build new ones) to verify the abstractions

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

Liskov Substitution Principle (LSP)

Why important?

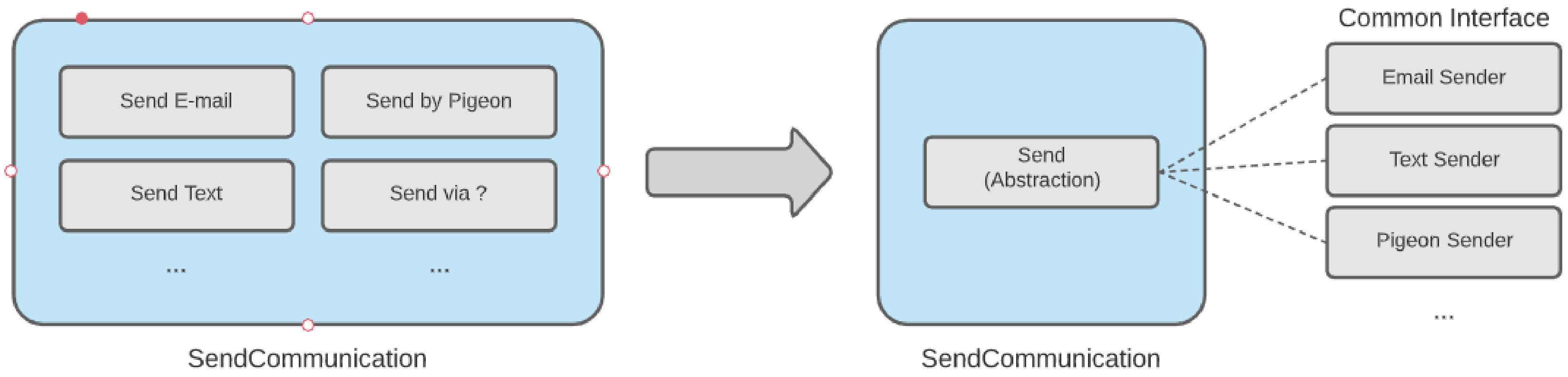
- Want to be able to use the abstractions created by OCP to extend existing functionality (vs. modify it)
- Especially useful when multiple variants of a type need to be processed as a single group
- Promotes looser coupling between our modules
- Without it, we may have to include if/else or switch blocks to route our logic
- Or keep adding parameters/properties for new but related types

Liskov Substitution Principle (LSP)

How to practice?

- Module members should reference the abstractions in consuming code
- Look for ways to encapsulate type-specific (or type-aware) logic in the type instead of in the code using the type
- Use existing tests (or build new ones) to verify our ability to effectively substitute

OCP & LSP

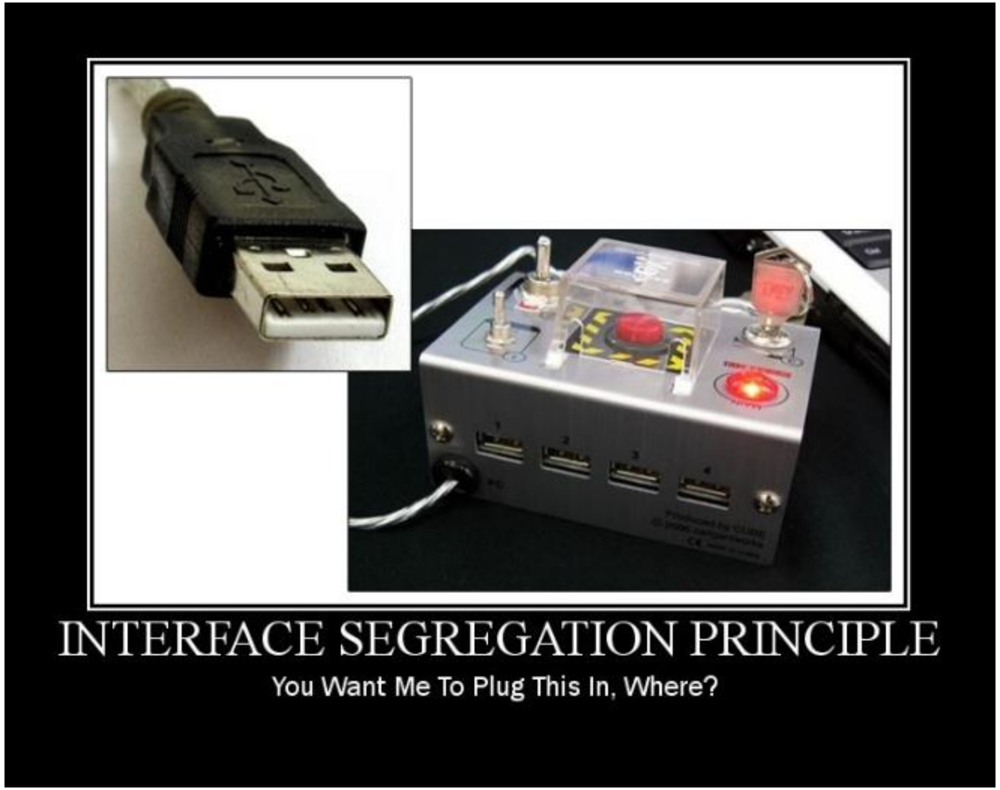

Send E-mail
Send by Pigeon
Send Text
Send via ?
...
...
SendCommunication
Send
(Abstraction)
SendCommunication
Common Interface
Email Sender
Text Sender
Pigeon Sender
...

Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use

Interface Segregation Principle (ISP)


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle (ISP)

Why important?

- By following SRP, OCP and LSP, we can build a set of cohesive abstractions enabling reuse in multiple clients
- However, sometimes the abstractions we need are not cohesive (even though they may seem like it at first)
- We need a mechanism for logical separation that still supports combining functions together in a loosely-coupled way
- Otherwise, we’ll see “bloating” in our abstractions that can cause unintended/unrelated impact during normal change

Interface Segregation Principle (ISP)

How to practice?

- In your abstractions, don’t force functions together that don’t belong together (or that you might want to use separately)
- Leverage delegation in the implementation of those abstractions to support variance
- Use OCP to bring together additional sets of features in a cohesive way (that still adheres to SOLID)
- Use existing tests (or build new ones) to verify aggregate features

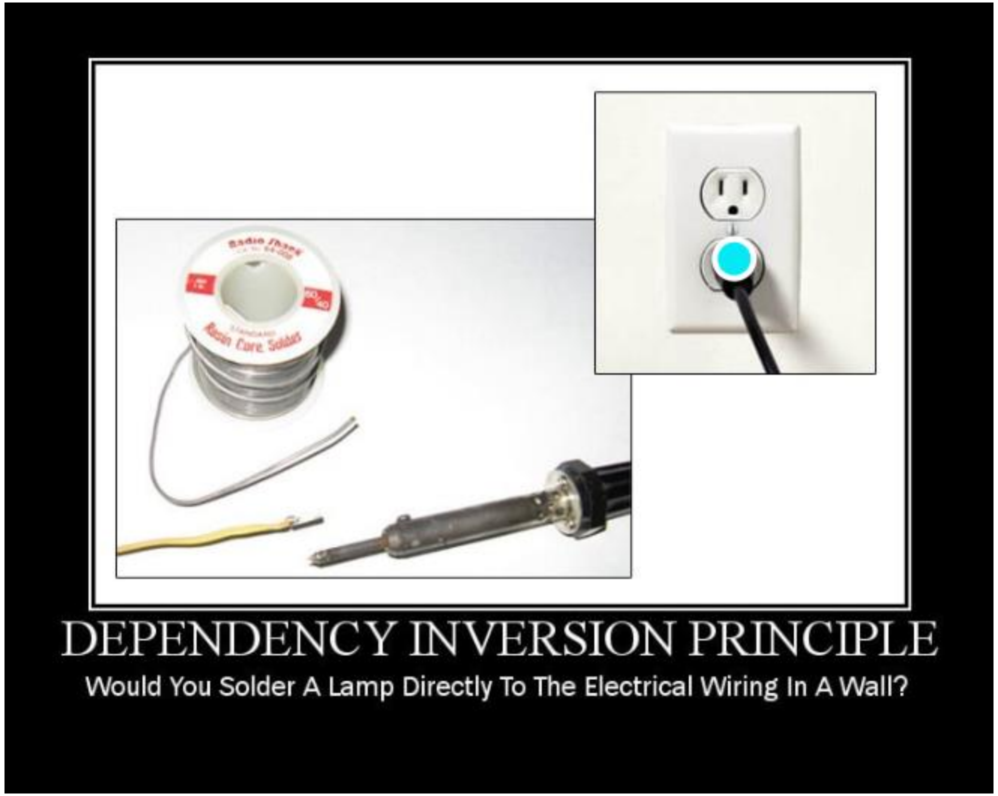
Dependency Inversion Principle (DIP)

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules – both should depend on abstractions

Abstractions should not depend upon details – details should depend upon abstractions

Dependency Inversion Principle (DIP)


Radio Shack
Rosin Core Solder
60/40
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle (DIP)

Why important?

- As with all the SOLID principles, we want to build reusable but loosely-coupled code modules
- We don’t want to limit reuse to lower-level utility classes only
- If higher-level modules are tightly-coupled to and dependent on low-level modules, change impact can cascade up
- The change impact can be transitive (flowing through multiple layers in-between)
- There are patterns and utility libraries built to enable

Dependency Inversion Principle (DIP)

How to practice?

- As with the other principles, use (and depend on) abstractions
- Use mechanisms like dependency injection and IoC (Inversion of Control) to build looser coupling between logic providers and consumers
- Build layering into your architectures and limit references to the same or immediately adjacent layer only
- Keep ownership of abstractions with the clients that use them or, even better, in a separate namespace/library
- Use existing tests (or build new ones) to verify functionality in each layer and use mocking techniques to isolate testing

SOLID Principles - Lab

Mechanics of Testing

Mechanics of Testing

For new requirements/user stories

- Tests should be structured to match the requirement
- Passing tests = satisfied requirement
- Edge cases / boundary conditions
- Exception cases (if applicable)
- Helps force us, as developers, to push for good requirements

Mechanics of Testing

What makes a requirement good?

- Unambiguous
- Testable/verifiable (good Acceptance Criteria)
- Clear, concise and precise
- Understandable and feasible
- Atomic and independent
- Consistent
- Implementation-free – not how to do but what to do

Impacts our ability to write a body of tests that are useful!

http://www.informit.com/articles/article.aspx?p=1152528&seqNum=4

Mechanics of Testing

For bug fixes

- Tests should be structured to expect the right answer
- Should initially fail (because of the bug)
- Passing tests = bug fix
- Use existing tests to confirm no regression from fix
- Use new tests as part of regression suite going forward

F.I.R.S.T.

The goal is clean code AND clean tests

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-Validating
- **T**imely

Clean Process

- Red, Green, Refactor
- Arrange, Act, Assert – testing state
- Mocking – testing behavior
- Be disciplined

Clean Tests

- One logical assertion – not necessarily only one Assert in code
- Avoid complexity where possible
- Keep stateless (or use isolated setup/teardown)
- Keep fast and narrow in scope – small bites
- Remember the tests are code too

Testing Public vs. Private

- Goal should be direct testing against public only
- Testing private/protected possible but not desirable
- Strive to exercise private through public tests
- Otherwise, can lead to internal coupling

Warning Signs

- Test runs that error/timeout inconsistently

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test

- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test

Warning Signs

- Test runs that error/timeout inconsistently

    Can signify instability/coupling in the tests or the code under test

- Test runs that are not repeatable

    Can signify unexpected/undesired statefulness in system under test

- Tests that are too complex to understand

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test
- Tests that are too complex to understand

  Can signify a deficiency in testability or design

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test
- Tests that are too complex to understand

  Can signify a deficiency in testability or design
- Test runs that are too slow

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test
- Tests that are too complex to understand

  Can signify a deficiency in testability or design
- Test runs that are too slow

  Can signify testing that is too coarse (functional vs. unit)

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test
- Tests that are too complex to understand

  Can signify a deficiency in testability or design
- Test runs that are too slow

  Can signify testing that is too coarse (functional vs. unit)
- Perpetually disabled tests

Warning Signs

- Test runs that error/timeout inconsistently

  Can signify instability/coupling in the tests or the code under test
- Test runs that are not repeatable

  Can signify unexpected/undesired statefulness in system under test
- Tests that are too complex to understand

  Can signify a deficiency in testability or design
- Test runs that are too slow

  Can signify testing that is too coarse (functional vs. unit)
- Perpetually disabled tests

  Can signify a deficiency in design or process

Principles

- Determine if the issue is with the test, the code under test or both
- Is there a problem with the test setup/teardown?
- If unit test, is there sufficient mocking in place?
- If an integration test, execute in separate, repeatable environment
- Ferret out inconsistency in your implementation

Principles, cont.

- Complex setup = complex code under test (simplify)
- Don’t over-engineer / don’t under-engineer – just enough
- Remove unnecessary or useless tests
- Need sufficient segregation between test projects
- Time is required to fix – prioritization

Python Testing Tools

unittest Module

- Included in Python standard library
- Provides a framework for building and running unit tests against Python code (functions, classes, etc.)
- A Python file houses code that will be used to automate the testing

unittest Module

- Test code imports unittest and module under test
- Test code includes a Python class that inherits from unittest.TestCase
- We write Python methods to exercise our business logic and verify expected outputs

unittest Module

- unittest includes multiple “assert” methods for checking different kinds of results (expected vs. actual)
- Can exercise tests through unittest framework
- Results will indicate success (or failure)

unittest Module

Options for executing tests

- *python -m unittest module_name*
- *python -m unittest module_name.TestClass*
- *python -m unittest module_name.TestClass.test_method*
- *python -m unittest path/to/test_file.py*

Can also use “discover” mode by running *python -m unittest*

unittest Module

- If test run fails, could be for 1 of a couple of reasons:
  - ➢ Code under test is incorrect
  - ➢ Test code is incorrect
- Verify the expected test results against your requirements
- If the expected result aligns, then fix the code under test

unittest Module

Assert methods include:

- assertEqual/assertNotEqual
- assertTrue/assertFalse
- assertIn/assertNotIn

https://docs.python.org/3/library/unittest.html#assert-methods

unittest Module – Setup and Teardown

Rather than repeat common logic across tests, unittest provides a couple of options for capturing:

- setUp
- tearDown
- Includes a couple of forms

unittest Module – Setup and Teardown

At the individual test level:

- setUp() – logic executed before each test is run; provides chance to setup common conditions
- tearDown() – logic executed after each test is run; provides chance to clean up (keep tests isolated)

unittest Module – Setup and Teardown

At the TestClass level:

- setUpClass() – logic executed before any tests in a class are run; requires decoration as a class method
- tearDownClass() – logic executed after all tests in a class are run; requires decoration as a class method

unittest Module – Skipping Tests

While permanently skipping tests can be a “smell”, unittest offers options for temporarily skipping:

- @unittest.skip() decorator
- @unittest.skipIf() decorator
- @unittest.skipUnless() decorator
- self.skipTest method – call within test to initiate a skip

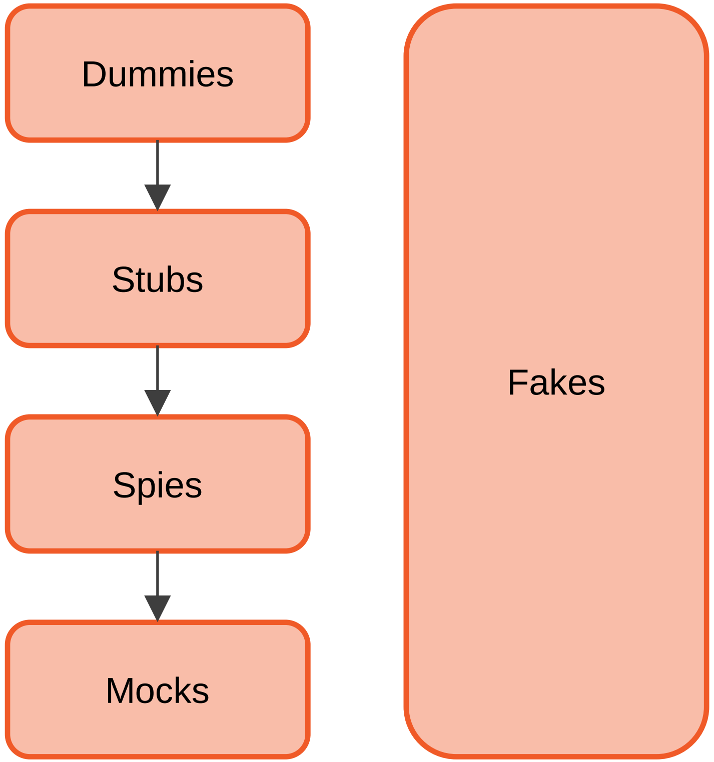
Mocking – Testing Components in Isolation

Mocking / Test Doubles

We use mocking/test doubles:

- To help isolate our testing to focused areas of code
- To manage dependencies and help control testing against them
- To force special cases that exercise specific code flows
- To bring stability and consistency to our tests
- Helps make the “unit” in unit testing possible

Mocking / Test Doubles



Dummies

- Simplest form of test double
- No real functionality – mostly a placeholder
- When dependency exists but is not needed for the test
- Create and use an empty or straightforward copy
- Returns meaningless value on every call

Stubs

- Same response regardless of parameters passed
- Used to test different paths through code
- Allows you to control the value of the dependency
- Can also use to force an exception
- Create a copy of the dependency and return same value

Spies

- Can return a value like a stub
- Also provides info about how called
  - Verify member call with specific parameter values
  - Verify member call a specific number of times
  - Verify whether member called or not
- Helpful with testing at third party boundaries
- Helpful with managing side effects of dependencies

Mocks

- Programmable spies
- Can return a set value like a spy
- Difference is ability to setup test data in the double
- Provides additional flexibility - can code the setup
- Still only returns set value but it’s the value coded
- Trade-off – requires setup work

Fakes

- Most powerful type of test double
- Acts like the class/component it’s doubling for
- Double that’s more sophisticated (but still controlled)
- Useful for managing downstream dependencies
  - ➢ Gating database connections
  - ➢ Gating web service calls
  - ➢ File operations
- Can appear to integrate but control results

Refactoring

Refactoring

Improving the internal structure of existing code without changing its external behavior or operation

Refactoring

If done correctly, clients of the code (either people or other code) should not see a difference

So, if that’s the case, why do it?

Refactoring – Why?

- Because we might not write the best code the first time
- Preventing “code rot” requires action
- We may not know what the code should look like until we have more of it in place
- Need an iterative process built into our development flow that allows us to focus on ongoing improvement

Refactoring – How?

- As we write code we look for “smells”
- Could include things like
  - Renaming variables/methods
  - Extracting methods or classes
  - Looking for code that can be removed due to previous refactoring efforts
  - Bringing code into adherence with SOLID
- Small steps (usually)
- Use automated tests to confirm no external change

https://refactoring.com/catalog/

Refactoring – Challenges?

- Lack of automated tests to help with verification
- Difficult when executing for the first time in a long time
- Fragile code/systems that break easily (see bullet 1)
- Difficult to quantify benefit of spend for it to the business

Refactoring – Overcoming the Challenges

- Build/improve tests as you go instead of letting any deficiencies in testing prevent
- You gotta start somewhere
- Work with managers to tie a cost to not doing – bug counts, regression impact, lost productivity
- Practice and pair to get good at it (use available tools)
- Measure before and after to demonstrate value

Coverage

Coverage

- Coverage in unit testing measures the % of your code under test that is exercised by a unit test
- Coverage value can range between 0 and 100 %
- Question: What is the right target % of coverage to aim for?

Coverage

- coverage.py can be used to measure test coverage in your Python app
- Install using pip (*pip install coverage*)
- Use *coverage run* to measure
- For unittest, use *coverage run -m unittest <target>*

Coverage – Measuring Results

- Use *coverage report* to report coverage results
- Alternatively, coverage html can be used to view coverage results in a more user-friendly manner

unittest, Mocking, and coverage - Demo

unittest, Mocking, and coverage - Lab

Metrics, Measurement, & Assessment

Quotes

“Every line is the perfect length if you don’t measure it.”
- Marty Rubin

Quotes

“Every line is the perfect length if you don’t measure it.”
- Marty Rubin

“What gets measured gets managed.”

- Pearl Zhu

Quotes

“Every line is the perfect length if you don’t measure it.”
- Marty Rubin

“What gets measured gets managed.”

- Pearl Zhu

“If you don’t collect any metrics, you’re flying blind. If you collect and focus on too many, they may be obstructing your field of view.”

- Scott M. Graffius

Quotes

“Every line is the perfect length if you don’t measure it.”
- Marty Rubin

“What gets measured gets managed.”

- Pearl Zhu

“If you don’t collect any metrics, you’re flying blind. If you collect and focus on too many, they may be obstructing your field of view.”

- Scott M. Graffius

“What science has failed to notice is that the measurement has become more real than the thing being measured.”

- R.A. Delmonico

Quotes

“That which cannot be measured cannot be proven.”
- Anthony W. Richardson

Quotes

“That which cannot be measured cannot be proven.”
- Anthony W. Richardson

“All conflict in the world is essentially about our differences in measurement.”

- Joseph Rain

Quotes

“That which cannot be measured cannot be proven.”
- Anthony W. Richardson

“All conflict in the world is essentially about our differences in measurement.”

- Joseph Rain

“It is impossible to escape the impression that people commonly use false standards of measurement – that they seek power, success and wealth for themselves and admire them in others, and that they underestimate what is of true value in life.”

- Sigmund Freud

TDD – What does success look like?

With TDD, ultimately, success looks like ongoing, improved quality in your software and systems

Not perfect quality (difficult to achieve) – but quality that gets progressively better and software that improves the business’s ability to achieve value

High-Level Goals

Goals:

- Improved customer satisfaction (both internal and external)
- Improved quality of product
- Improved time-to-market of product
- Improvements in software design process
- Improvements in software development process

Metrics for assessing quality

Escaped bugs – A measure of bugs detected (and reported) by customers

Provides a real-world view of the product quality – from the outside in

Metrics for assessing quality

Defect Removal Efficiency (DRE) – A measure of team efficiency at bug removal before customer is affected

Helps assess ability of the team to stay ahead of issues

Metrics for assessing quality

New Bugs vs. Closed Bugs – A measure of new bugs being opened vs. existing bugs being closed

Helps assess if system/product is moving in the right direction in terms of code quality

Metrics for assessing quality

Bug Burn Down – A measure of open bugs and projected closure of existing bugs based on average rates

Helps assess team efficiency at reducing the bug count and identify blockers that are preventing progress

Metrics for assessing quality

% of High/Critical Priority (or Severity) Bugs – A measure of % of bugs at high/critical priorities (or severities) against total number

Helps assess critical quality areas that might need additional team focus while also informing the use of bug count as a metric/measure

Metrics for assessing quality

Defect Density – A measure of number of bugs per LoC or KLoC

Helps assess potentially unstable areas of the code or areas with increased complexity relative to others

Metrics for assessing quality

Code Churn – A measure of the number of lines of code deleted, added or changed in code checked in to your repository

Helps assess areas of the code requiring more “care and feeding” over time which can imply instability, balanced against any ongoing refactoring efforts

Metrics for assessing quality

Code Coverage – A measure of the % of lines of code executed or exercised during an execution of your test suite

Helps assess the overall testing process – i.e. are all (or a sufficiently significant number) of your lines of code being exercised via test?

Can also be paired with Static Code Analysis

Metrics for assessing quality

Release Confidence – A qualitative measure of level of confidence the team has in a pending release of the product

Helps provide a team “gut check” as to readiness to release or expected stability of product on release

TDD’s Place

- The principles of TDD and automated testing are all about improving software quality
- Starts with designing for testability
- Targets clean code, clean design and clean tests throughout the development effort
- In general, software development efforts that include automated testing, exhibit the potential to:
  - ➢ Help an organization achieve its quality goals
  - ➢ Help an organization drive key metrics in the right direction
- However, it takes time, effort and commitment

Thank you!

If you have additional questions,
please reach out to me at:
(asanders@gamuttechnologysvcs.com)

DevelopIntelligence
A PLURALSIGHT COMPANY