A

AAA pattern (aka 3A, Triple A, Given-When-Then) (Ch 3)

- Provides simple, uniform structure, which is the biggest advantages of the pattern bc as you get used to it, can easily read & understand any test
    - Arrange: bring the system under test (SUT) & dependencies to a desired state (initialize variables)
    - Act: call methods on the SUT, pass prepared dependencies, capture the output value (if any)
    - Assert: verify the outcome, which may be represented by the return value, final state of the SUT & its collaborators, or methods the SUT called on those collaborators
- aka **Given, When Then**
    - Given: Arrange
    - When: Act
    - Then: Assert

Access modifiers (Ch 5)

- `private, public, internal` (only 3 in `C++`)
- Keywords in OOP (Object-Oriented Programming) languages that set the accessibility of classes, methods & other members
- A specific part of programming language syntax used to facilitate the encapsulation of components

Active Record Pattern (Ch 7)

- An approach in which a domain class retrieves & persists itself to the database
- Works fine in simple & short-lived projects, but often fails to scale as the code base grows due to lack of separation between 2 responsibilities: business logic & communication with out-of-process dependencies

Adapters (Ch 9)

- Abstract non-essential technical details of 3rd-party code
- Define the relationship with the library in your application's terms
- *Act as an anti-corruption layer between your code & the external world,* helping you:
    - Abstract the underlying library's complexity
    - Only expose features you need from the library

    - Do that using your project's domain language

Aggregate Pattern (Ch 7)
- From *Domain-Driven Design* by Eric Evans
- Goal: reduce connectivity between classes by grouping them into *clusters (aggregates)*
    - The classes are highly connected inside those clusters, but the clusters themselves are loosely-coupled
    - Such a structure *decreases the total number of communications* in the code base
    - The reduced connectivity, in turn, *improves testability*

Anti-pattern (Ch 11)
- Common solution to a recurring problem that looks appropriate on the surface but leads to problems further down the road
- Additional reading
    - Anti-pattern: Lack of Documentation and Comments
    - Anti-pattern: Magic Numbers

Assertion-free testing (Ch 1)
- Tests without assertion statements
    - If they don't have assertions, then they don't verify anything
    - You might see this if teams are using code coverage as a requirement & intend to get 100% code coverage, but:
        - Using code coverage as a requirement can lead to artificial targets: bad/false positive, which is a liability & hides bugs

Atomic Updates (Ch 10)
- Executed in *all-or-nothing manner*
- Each update in the set of atomic updates must either **be complete in its entirety or have no effect whatsoever**
- Related concept: ACID Principles for Databases in which the `A` stands for atomicity

B

Bad Positive / Good Negative (Ch 1)
- Terms have roots in statistics, but can be used to describe the quality of unit tests

- Bad Positive: Your test exists, you have coverage, but doesn't necessarily mean that your tests are effective or useful
    - **False Positive** (Ch 4):
        - A false alarm
        - A result indicating that the test fails, whereas the functionality it covers works as intended
        - Can have a devastating effect of the test suite
            - Dilute your ability & willingness to react to problems in code bc you get accustomed to false alarms & stop paying attention to them
            - Diminish your perception of tests as a reliable safety net & lead to losing trust in the test suite
        - Result of tight-coupling between tests & internal implementation details of the SUT
            - To avoid, the test must *verify the end result* the SUT produces, *not the steps it took to do that*
        - Don't have much of a negative effect in the beginning of the project, but they become increasingly important as the project grows: as important as false negatives (*unnoticed bugs*, Type II error)
- **Good negative**: A process that reveals your code is bad is ultimately good / successful in that it yields useful information
    - These phrases were introduced while discussing the limits of Code Coverage programs
        - "The ability to test code is a good litmus test, but only in 1 direction"
            - "Good litmus test"
                - *A decisively indicative test*
                - *Can be used to make a judgment about whether someone or something is acceptable or not*
            - If you can't test your code, might be bad code
            - If you can test your code, that doesn't guarantee quality
    - **True Positive** (Ch 4)
        - *"When the functionality is broken & the test fails, it's also a correct inference, bc you expect to see the test fail when the functionality is not working properly, which is the whole point of unit testing"(76).*
        - When the test doesn't catch an error, that's a problem

- *"In the context of testing, positive means that some set of the conditions is now true"* (77).

	Functionality *Correct*	Functionality *Broken*
Test *passes*	Correct inference (true negatives)	Type II error (false negative) *Protection Against Regressions
Test *fails*	Type I error (false positive) *Resistance to Refactoring	Correct Inference (true positives)

Black-box (functional, data-driven) **Testing** (Ch 4)

- Examines the functionality of a system without knowing its internal structure
- Normally built around specifications & requirements
  - *What* the app is supposed to do rather than *how* it does it
  - **White-box (Clear-box) Testing**
    - The opposite
    - Verifies the app's inner workings
    - Tests are derived from the source code, not requirements or specifications

	Protection against regressions	Resistance to refactoring
White-box Testing	Good	Bad
Black-box Testing	Bad	Good

Branch Coverage (Ch 1)

- Type of code coverage metric
- More precise by helping cope with code coverage's shortcomings

  Branch coverage =

  Branches traversed / Total number of branches

Brittle Test (Ch 4)

- Test that runs fast & has a good chance of catching a regression but does so with a lot of false positives
- Can't withstand a refactoring & will turn red regardless of whether the underlying functionality is broken

C

CanExecute/Execute Pattern (Ch 7)

- Helps avoid leaking of business logic from the domain model to controllers
- Helps you to consolidate all decisions in the domain layer
- Prepare the data, make a decision, acts on the decision

CAP Theorem (Ch 4)

- It's impossible for a distributed data store to simultaneously provide more than two of the three guarantees:
    - **Consistency**
        - Every read receives the most recent write or an error
    - **Availability**
        - Every request receives a response (apart from outages that affect all nodes in the system)
    - **Partition tolerance**
        - The system continues to operate despite network partitioning (losing connection between network nodes)
- There is always a trade-off between consistency & availability that changes with different contexts

Circular Dependency (Ch 8)

- aka cyclic dependency
- 2 or more classes that directly or indirectly depend on each other to function properly

Classical (*Detroit*) School (Ch 2)

- "Classical" bc how everyone originally approached unit testing & TDD (test-driven-development)
- Unit tests isolated from each other
- Unit of behavior, <u>not</u> unit of code
- *Only shared dependencies* should be replaced with test doubles
    - Shared dependencies affect other's flow

Client (Ch 5)

- Overloaded, can mean different things depending on where the code resides
- Common examples: client code from same code base, an external app, or user interface

Code Complexity (Ch 7)

- The number of decision-making (branching) points in the code
    - The greater this number, the higher the complexity
- Cyclomatic Complexity
    - Computer Science term
    - Indicates the number of branches in a given program or method, calculated by
        - 1 + <number of branching points>
    - Method with no control flow statements (if statements / conditional loops) has cyclomatic complexity of 1 + 0 = 2
    - Can think of it in terms of the *number of independent paths through the method from an entry to an exit*, or the number of tests needed to get 100% branch coverage
        - Number of branching points is counted as the number of simplest predicates involved
        - A statement like IF condition1 AND condition2 THEN ... is equivalent to IF condition1 THEN IF condition2 THEN...
            - complexity would be 1 + 2 = 3
- We're interested in the separation of business logic & orchestration
    - These 2 responsibilities can be analyzed through code depth vs code width
        - Your code can either be deep (complex or important) or wide (work with many collaborators), but never both

Code Pollution (Ch 6, 11)

- *Anti-pattern* that mixes test & prod code, increases maintenance costs
- Process of adding code whose sole purpose is to *enable or simplify unit testing*
    - A pattern that highlights the complexity of a code in specific unit testing
    - Takes place when devs include more code in their DB to make unit testing work

- Solution: *Do not introduce additional code production*

Communication-Based Testing (Ch 6)
- Uses mocks to verify communications between the system under test & its collaborators
- **Output-Based Testing**
  - Verify the output the system generates
- **State-Based Testing**
  - Verify state of the system after an operation is complete

	Output-Based	State-Based	Communication-Based
Due diligence to maintain resistance to refactoring	Low	Medium	Medium
Maintainability costs	Low	Medium	High

Concrete Class (Ch 8)
- A *class that we can create an instance of, using the new keyword*
- An abstract class can be instantiated either by a concrete subclass or by defining all the abstract method along with the new statement

Controllers (Ch 7)
- Code that doesn't do complex / business critical work by itself, but coordinates the work of other components like domain classes & external applications
  - Overcomplicated Code
    - Lots of collaborators, complex & important, like a fat controller
      - Controllers that don't delegate complex work anywhere & do everything themselves

Cost component (Ch 1)
- A metric that describes how much labor is required for maintenance, usually time
- *How much time spent o*n:

- Refactoring test & underlying code
- Running test on each change
- Dealing with false alarms raised by the test
- Spending time reading the test when you're trying to understand how the underlying code behaves

Coverage metric (Ch 1)

- Shows how much source code a test suite executes (none/0%-100%)
  Code coverage (test coverage) =
  Lines of code executed / Total number of lines

CQS (Command Query Separation) Principle (Ch 5)

- Every method should be either be a command or a query, but not both
  - Commands are methods that produce side effects & don't return any value (return void)
    - Side effects: mutating an object's state, changing file in the file system
  - Queries are the opposite
    - Side-effect free
    - Return a value
- Test doubles that substitute commands are mocks
- Test doubles that substitute queries are stubs

D

Database Schema (Ch 10)

- Tables, views, indexes, stored procedures & anything else that forms a blueprint of how the DB is constructed
- Reference Data:
  - Part of the DB Schema
  - Data that must be pre-populated in order for the app to operate properly
  - Regular Data: your app *can modify that data*, while *reference data is the opposite*

Data Motion (Ch 10)

- The process of changing the shape of existing data so that it conforms to the new database schema

Dead Code (Ch 11)

- Code that's not being used
- Extraneous code left after a refactoring - delete if possible
- **Code Graveyard**: Dead project that no one has proper knowledge to fix
    - A project is dead when there is no intent on reviving it

Diagnostic Logging (Ch 8)

- Helps devs understand what's going on inside the app
- Support Logging produces messages that are intended to be tracked by support staff or system administrators
- Structured Logging is a logging technique where capturing log data is decoupled from the rendering of that data

Domain Event (Ch 7)

- Describes an event in the application that is meaningful to domain experts
    - The meaningfulness for domain experts is what differentiates domain events from regular events (such as button clicks)
- These are often used to inform external applications about important changes that have happened in your system
- Should always be named in the past tense bc they represent things that already happened
- These are values - they're immutable & interchangeable

Domain Model (Ch 8)

- The collection of domain knowledge about the problem your project is meant to solve
- *A domain model is a conceptual model of the domain that incorporates both behavior and data.*

Domain Significance (Ch 7)

- Shows how significant the code is for the problem domain of your project
- Normally, all code in the domain layer has a direct connection to the end users' goals & thus exhibits a high domain significance while utility code doesn't have a connection

E

Encapsulation (Ch 5)
- Act of protecting code against invariant violations
    - **Invariant**: a condition that should be held true at all times

Enterprise application (Ch 1)
- App that aims at automating or assisting an org's inner processes
- Can take many forms, but usual characteristics include:
    - High business logic complexity
    - Long project lifespan
    - Moderate amounts of data
    - Low or moderate performance requirements

F-G

Fail Fast Principle (Ch 8)
- Advocates for making bugs manifest themselves quickly & is a viable alternative to integration testing
- Process of stopping current operation as soon as any unexpected error occurs
- Makes your app more stable by:
    - Shortening the Feedback Loop
        - Sooner you detect a bug, the easier it is to fix
        - A bug that's already in prod is orders of magnitude more expensive to fix compared to a bug found during development
    - Protecting the Persistence State
        - Bugs lead to corruption of the app's state
        - Once that state penetrates into the DB, it becomes much harder to fix
        - Failing fast helps you prevent the corruption from spreading
- Stopping the current operation is normally done by throwing exceptions bc exceptions have semantics that are perfectly suited for the FFP: they interrupt the program flow & pop-up to the highest level

of the execution stack, where you can log them & shut down or restart the operation
- A failing precondition signifies an incorrect assumption made about the app state, which is always a bug
- Reading data from a config file - can arrange the reading logic such that it will throw an exception if the data in the config is incomplete or incorrect; can also put this logic close to the app startup, *so that the app doesn't launch if there's a problem with it's configuration*

Functional Programming (Ch 6)
- Programming with mathematical functions (aka pure function)
    - Function or method that doesn't have any hidden inputs or outputs
        - *Side effects* (most prevalent type)
            - An output that isn't expressed in the method signature, therefore, hidden
            - An operation creates a side effect when it mutates the state of a class instance, updates a file on the disk, etc
        - *Exceptions*
            - When a method throws an exception, it creates a path in the program flow that bypasses the contract established by the method's signature
            - The thrown exception can be taught anywhere in the call stack, thus introducing an additional output that the method signature doesn't convey
        - *A reference to an internal or external state is a hidden output*
            - Method that gets current time & time
            - Can query data from the db, refer to a private mutable field
            - These are all inputs to the exception flow that aren't present in the method signature & therefore, hidden
    - All inputs & outputs of a mathematical function must be explicitly expressed in its method signature (method's name, arguments & return type)
    - Produces the same output for a given input regardless of how many times it is called
    - Being explicit makes them extremely testable
    - In mathematics, a function:

        - Relationship between 2 sets that for each element in the 1st set, finds exactly 1 element in the 2nd set
- **Functional Architecture** (hexagonal architecture taken to an extreme):
    - Maximizes the amount of code written in a *purely functional (immutable)* way, while minimizing code that deals with side effects
        - Immutable:
            - Unchangeable
            - Once an object is created, its state can't be modified
        - Mutable (the opposite):
            - Changeable object
            - Can be modified after it is created
    - Divides all code into function core & mutable shell
        - Function Core makes decisions
        - Mutable Shell supplies input data to the functional core, converts decisions the core makes into side effects

H

Happy Path (Ch 8)
- Successful execution of a business scenario
- Edge Case: When the business scenario execution results in an error

Hexagonal architecture (Ch 5)
- Set of interacting apps represented as hexagons
- Each hexagon consists of 2 layers: domain & app services
- Each layer in a hexagon exhibits observable behavior & contains its own set of implementation details
- Emphasizes separation of concerns between layers, 1-way flow of dependencies from app to domain layer & no direct access to the domain layer

Humble Object Design Pattern (Ch 7)
- Introduced by Gerard Meszaros in *xUnit Test Patterns: Refactoring Test Code*
- 1 of the ways to battle code coupling

    - Process of decoupling code from a difficult dependency by bringing logic of this code under test & extracting testable part out of it
        - The result is the code becoming a *thin, humble wrapper* around that testable part: it glues the hard-to-test dependency & the newly extracted component together, but itself contains little or no logic, thus doesn't need to be tested
- A way to implement the Single Responsibility Principle
        - Each class should have only 1 single responsibility - one such responsibility is always business logic; this pattern can be applied to segregate the logic from pretty much anything
        - *"...every class should have only one reason to change"*

I-L

Incoming Interactions (Ch 5)
- Calls the SUT makes to its dependencies to get input data
- Outcoming Interactions
    - Calls from the SUT to its dependencies *that change the state* of those dependencies

Integration Test (Ch 2, 8)
- Verifies 2 or more units of behavior
    - Often a result of trying to optimize speed
    - Plays a significant part in contributing to software quality by verifying the system as a whole
    - Verifies that your *code works in integration with shared dependencies, out-of-process dependencies, or code developed by other teams in the organization*
    - Provide better protection against regressions & resistance to refactoring while unit tests have better maintainability & feedback speed
    - **End-to-end (E2E)** tests are a subset of integration tests
        - Includes more out-of-process dependencies
        - Test verifies the system from the end user's POV, including all the external apps this system integrates with

    - There may be no test version of some dependencies, or it may be impossible to bring those dependencies to the required state automatically
    - May need to still use a test double, meaning there isn't a distinct line between integration & E2E tests

Inter-system Communication (Ch 5)
- Communication between app & external apps
- Part of observable behavior, with the exception of external systems that are accessible only thru your app, but also implementation details bc the resulting side effects aren't observed externally
  Intra-system Communication
    - Communication between classes inside app
    - Are implementation details

M-N

Managed Dependencies (Ch 8)
- Out-of-process that are *only accessible through your app*
- Interactions with them *aren't observable externally*
- Typical example is app database: external systems don't access your DB directly, but do that through your API your app provides
- Unmanaged Dependencies
    - The opposite; other apps have access to it; *you don't have full control over these*
    - Interactions are observable externally / are visible to the external world
    - Example: SMTP server, message bus
        - Both *produce side effects visible to other apps*

Message Bus (Ch 8)
- aka Service Bus, provides a way for one (or more) application to communicate messages to one or more other applications
- An unmanaged dependency bc sole purpose is to enable communications with other systems

Mock (Ch 2, 9)

- Subset of test doubles, used synonymously although they're technically not
    - *Test double is general, overarching term* that describes all kind of non-production-ready, fake dependencies in a test while mock is just 1 kind of such dependencies
- *Special kind of test double that allows you to examine interactions between the SUT & its collaborators*

Mock chain (Ch 6)
- Mocks or stubs returning other mocks, which also return mocks & so on, several layers deep

Mockist (*London*) School (Ch 2)
- "London" bc geographical location of popularity
- Units should be isolated
- Unit is usually a class
- All dependencies (except immutables) should be test doubles in tests
- Benefits
    - Better granularity
    - Ease of testing interconnected classes
    - Ease of debugging
- Issues
    - Focus on classes misplaced, as tests should verify units of behavior & not exactly units of code & test doubles doesn't fix this problem (just hides it)
    - Ease of debugging might not be significant if your process is just about what you edited last
    - Overspecificiation & coupling to the **SUT(*system under test*)** implementation details

Model Database (Ch 10)
- A dedicated DB instance
- Serves as a reference point
- During development, all schema changes accumulated in this instance
- Upon production deployments, compare the prod & model databases, use a special tool to generate upgrade scripts & run those scripts in prod

Model-View-Presenter Design Pattern, (MVP), Model-View-Controller Design Pattern (MVC) (Ch 7)

- Help decouple business logic (*Model*), UI concerns (*View*) & the coordination between them (*Presenter* or *Controller*)
    - The Presenter & Controller components *are humble objects*
        - They glue the View & the Model together

MUT (*Method Under Test*) / SUT (*System Under Test*)

- Method in the SUT called by the test
- MUT/SUT often synonyms, but *normally MUT refers to a method while SUT refers to the whole class*

O-Q

Object graph (Ch 2)

- The web of communicating classes solving the same problem
- This web might become quite complicated: every class in it may have several immediate dependencies, each of which relies on dependencies of their own & so on, or
    - **Circular dependencies**
        - Where the chain of dependency eventually comes back to where it started

Object Mother Pattern (Ch 10)

- A class or method that helps create test fixtures (objects that test runs against)
- **Test Builder Pattern**
    - Helps achieve the same goal of reusing code in arrange sections
    - Exposes a fluent interface instead of plain methods
    - Can improve readability, but requires boilerplate

Open-Closed Principle (OCP) (Ch 8)

- SOLID Principle
- Objects or entities should be open for extension but closed for modification
- A class should be extendable without modifying the class itself

- Can extend a class' behavior without changing the source code itself

Operation (Ch 5)
- A method that performs a calculation or incurs a side effect

Out-of-process dependency (Ch 2)
- *Runs outside the app's execution process*
- Proxy to data that is not yet in the memory
- Corresponds to shared dependency in vast majority of cases, but not always, a db (database) is both out-of-process & shared, while read-only db is also out-of-process but not shared
- Tests can't mutate data in such a db & can't affect each other's outcome

Overspecification (Ch 5)
- Practice of verifying things that aren't part of the end result
- Commonly take place when examining interactions

Parameterized tests (Ch 2)
- Type of data-driven testing that allows you to execute the same test, multiple times using different parameters
- Help reduce amount of code needed for similar tests
- Drawback: sacrifice readability as they become more generic

R

Read Operation (Ch 10)
- Example: Returning user info to the external client
- Write Operation
    - An operation that leaves a side effect in the DB & other out-of-process dependencies
    - Example: Changing a user email

Refactoring (Ch 4)
- Changing existing code without modifying its observable behavior
- Intention is usually improvement to code's nonfunctional characteristics
    - Increase readability
    - Reduce complexity

- *Examples: renaming a method, extracting a piece of code into a new class*

Referential Transparency (Ch 6)
- The ability to replace a method call with the corresponding value
- If a method call can be replaced, it's likely a mathematical function

Regression (Ch 1, 4)
- aka software bug (synonymous, can be used interchangeably)
- When a feature stops working as intended after a certain event (usually a code modification, roll-out new functionality)
- The author is using this as a general, umbrella term to reference bugs, they're not referring to regression testing:
    - A type of software testing to confirm that a recent program or code change has not adversely affected existing features
    - A full or partial selection of already executed test cases that are re-executed to ensure existing functionalities work fine
    - Done to ensure that new code changes do not have side effects on the existing functionalities
    - Ensures that the old code still works once the latest code changes are done
    - Includes re-running functional & non-functional tests to ensure that previously developed & tested software still performs as expected after a change
        - If not, that would be called *a regression*

Repository (Ch 10)
- A class that enable access to & modification of the data in the DB
- Lifecycle: short-lived; can dispose of a repository as soon as the call to the DB is completed; always work on top of the current transaction
- When connecting to the DB, a repository enlists itself into the transaction so that any data modifications made during that connection can later be rolled back by the transaction

S

Shared dependency (Ch 2)
- Shared *between tests* & provides means for those tests to affect each other's outcome, like a static mutable field or database/db

- A change to a mutable field is visible across all unit tests running within the same process
- While a private dependency is not shared

Signal-to-noise ratio (Ch 4)
- Signal is the number of bugs found
- Noise is the number of false alarms raised
- Test accuracy = Signal / Noise

Single Responsibility Principle (Ch 11)
- SOLID Principles of OOP
- *Every class, module, or function in a program should have 1 responsibility/purpose in a program*
- As a commonly used definition, "*every class should have only one reason to change*"

Software entropy (Ch 1)
- Phenomenon of *quickly decreasing development speed*
- Process in which code base becomes unreliable
    - Without proper care (cleaning, refactoring)
    - Complexity, disorganization
    - Bugs that introduce other bugs, domino-effect type of break
    - **Entropy**
        - A metric that describes the amount of disorder in a system
        - Mathematical & scientific concept that can also be applied to software systems

Spy (Ch 9)
- aka *handwritten mocks*
- Variation of a test double that serves the same purpose as a mock
- Only difference is that spies are written manually, while mocks come from frameworks

State (Ch 5)
- Current condition of the system

T

Tautology Tests (Ch 4, 9)

- A tautology is "a statement that is true by necessity or by virtue of its logical form"
- The most basic form of tautological test is when the test & the code use the same formula
- *Don't test/verify anything* bc they're set-up in such a way that they always pass or contain *semantically* meaningless assertions
    - In a way that is connected with the meaning of words; grammatically correct / syntax correct, but meaning is nonsensical

Test Double (Ch 2)

- Object that looks & behaves like its release-intended counterpart but is actually a simplified version that reduces the complexity & facilitates testing
- Term 1st introduced by Gerard Meszarose (*xUnit Test Patterns*)
- Name itself comes from the notion of a stunt double in movies

Test-Driven Development (TDD)

- Software development process that relies on tests to drive the project development
- Consists of 3 stages repeated for each case:
    1. Write a failing test
        - To indicate which functionality needs to be added & how it should be behave
    2. Make the tests pass
        - Doesn't have to be elegant or clean, but just enough to pass the tests you just wrote
    3. Refactor
        - Under protection of passing test, can safely clean-up for readability/maintainability

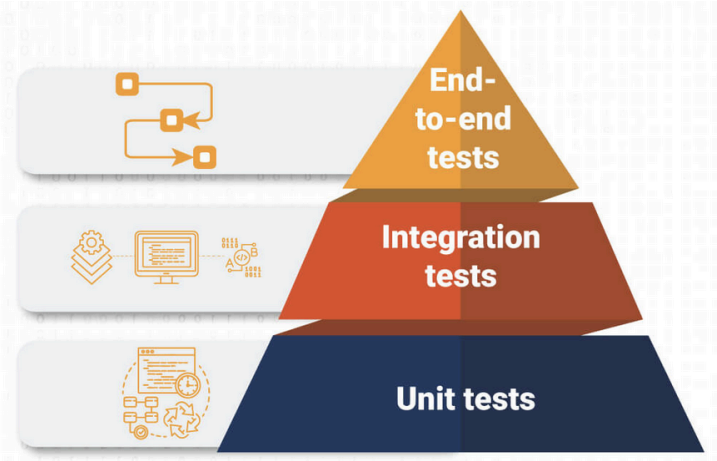
Text fixture (Ch 3)

- *An object the test runs against*
    - *Can be data that remains in a known, fixed state* before each test run & *produces the same result, hence "fixture"*

    - Can be a regular dependency, an argument passed to the SUT
- Attribute from NUnit testing framework that marks classes continuing tests

Test isolation (Ch 5)
- Ability for tests to run in parallel, sequentially & in any order

Test Pyramid (Ch 4)
- Concept that advocates for a certain ratio of different types of tests in the test suite
- The width of the pyramid layers refers to the prevalence of a particular type of test
- Test count is widest at bottom / horizontal, emulating user increases vertically
    - *"The wider the layer, the greater the test count. The height of the layer is a measure of how close these tests are to emulating the end user's behavior"* (88).



Transaction (Ch 10)
- A class that either commits or rolls back data updates in full
- Will be a custom class relying on the underlying DB's transactions to provide atomicity of data modification
- Lifecycle: Lives during the whole business operation & is disposed of at the very end of it

Trivial Test (Ch 4, 7)
- Maximizes 2 out of 3 attributes at the expense of the 3rd
    - Runs very quickly, low chance of producing a false positive / good resistance to refactoring, unlikely to reveal any

regressions bc there's not much room for a mistake in the underlying code
- Covers a simple piece of code, something unlikely to break bc it's too trivial
- Low complexity & domain significance, few collaborators - not worth testing at all
    - Examples: parameterless constructors, 1-line properties

U-Z

Unit of Work Pattern (Ch 10)
- Maintains a list of objects affected by a business operation
- Once the operation is completed, the unit of work figures our all updates that need to be done to alter the DB & executes those updates as a single unit

Unit Test (Ch 1)
- Verifies a single unit of behavior, quickly & in isolation from other tests

Value object (Ch 6)
- A class whose instances are compared by value & not by reference

Volatile dependency (Ch 2)
- Exhibits 1 of the following:
    - *Introduces requirements* to set-up & config runtime environment in addition to what's installed on a developer's machine by default (dbs, APIs)
    - *Contains nondeterministic behavior* (random number generator, class returning timestamp)

YAGNI (*You Aren't Going To Need It*) Principle (Ch 8)
- Advocates against investing time in functionality that's not needed right now
- You shouldn't develop this functionality, not should you modify your existing code to account for the appearance of such functionality in the future
    - *Opportunity Cost*

    - If you spend time on a feature that the business doesn't need now, you're taking away from features needed now -> wasteful
    - It's more beneficial to implement the functionality from scratch when the actual need for it emerges
- *The less code in the project, the better*
    - Introducing code just in case without immediate need increases your code base's cost of ownership -> better to postpone introducing new functionality until as late a stage of your project as possible