Unit Testing

Sam Huffer, 31 October 2018

Unit what?

Unit testing is a software development process where the individual units of a program are tested, usually by the software developers themselves, before any other type of testing is conducted. Units usually have one or a few inputs, and usually a single output. In object-oriented programming, methods are the smallest units and thus are the ones tested (STF).

A number of tasks are involved in Unit Testing (STF):

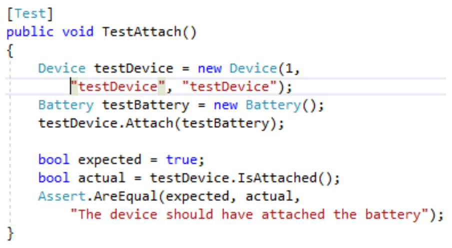
1) Planning which units in the program will be tested, and how those units will be tested. Only the characteristics that are vital to the performance of a unit should be tested (Rouse).
2) Writing the Unit Tests for each of the units to be tested, specifying inputs for the unit and the output it is expected to generate as a consequence.
3) Writing the code for the units that will be tested.
4) Running the Unit Tests for each unit. At this point, a number of units may fail their tests. The output of the unit tests should provide some indication of what problems that unit has run into, giving developers an idea of how to change the unit's code so that it passes its tests.
5) Modifying the code for the units that failed their tests, rewriting the erroneous code, or refactoring the poorly structured code, that caused the unit to fail its test, with the intention of having the altered unit pass its Unit Test (Rouse).
6) Re-running the Unit Tests. If a unit passes, move onto modifying the next unit that has failed. If it has not passed, keep modifying it until it passes.

Once all units being tested have passed their unit tests, developers can move onto conducting other kinds of testing, or adding more features which will require unit testing in turn.

What's the catch?

Disadvantages of unit testing include:

- Unit testing can have a steep learning curve. If the development team has not been educated about unit testing, it will need to learn what unit testing is, how to do it, and how to use automated tools to facilitate it on an ongoing basis (Rouse).
- It takes more up-front time and effort to conduct unit testing than it would take to just write the code and do informal testing.
- Developers may be required to use a version control system if they wish to conduct unit testing (Set Skill).



```
[Test]
public void TestAttach()
{
    Device testDevice = new Device(1,
        "testDevice", "testDevice");
    Battery testBattery = new Battery();
    testDevice.Attach(testBattery);

    bool expected = true;
    bool actual = testDevice.IsAttached();
    Assert.AreEqual(expected, actual,
        "The device should have attached the battery");
}
```

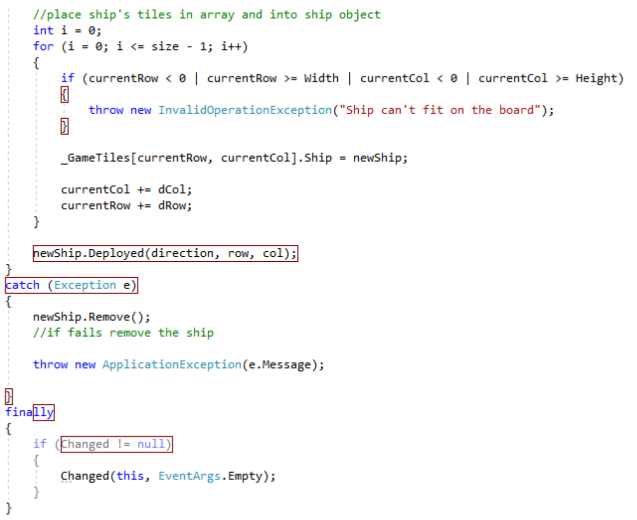
Figure 1: A unit test for Device.Attach() *from the rover simulator task of the unit* Object Oriented Programming*. It shows objectively what that method needs to accomplish and whether it does it properly, not relying on the developer's vague feeling or inaccurate opinion that they've gotten this section of code right when creating it or modifying it later.*

However, its advantages substantially outweigh its disadvantages (STP):

- Unit testing is much more reliable and objective than informal developer testing (see fig. 1).
- It increases confidence in changes to code, as any time the code is altered, any bugs introduced will be

promptly caught and their location more quickly identified, and thus fixed more quickly (see fig. 1).

- It makes debugging easier, as developers only need to look at the latest changes if a test fails (Set Skill).
- Unit testing reduces the overall cost of fixing bugs by identifying problems earlier in development:
  - If a bug is found early in development due to unit testing, it can be fixed quickly and easily, saving more time in testing than it takes to set up the unit testing.
  - If a bug is not found until late in the process, it can take a lot more time and effort to find and fix it, especially since developers that don't use unit testing will often need to burrow down into the code with break points to locate the problem (for example, see fig. 2).
  - The earlier in development a bug is identified and fixed, the less impact it will have on other existing code (due to there being less code), thus lowering the chance it will break something else and create another bug to be fixed, and therefore the time spent fixing such additional bugs.
  - The earlier bugs are identified, the less likely the developer will have to deal with compound errors (i.e. errors that don't seem to break anything initially, but conflict with something further down the line) (Rouse).



```
    //place ship's tiles in array and into ship object
    int i = 0;
    for (i = 0; i <= size - 1; i++)
    {
        if (currentRow < 0 | currentRow >= Width | currentCol < 0 | currentCol >= Height)
        {
            throw new InvalidOperationException("Ship can't fit on the board");
        }

        _GameTiles[currentRow, currentCol].Ship = newShip;

        currentCol += dCol;
        currentRow += dRow;
    }

    newShip.Deployed(direction, row, col);
}
catch (Exception e)
{
    newShip.Remove();
    //if fails remove the ship

    throw new ApplicationException(e.Message);

}
finally
{
    if (Changed != null)
    {
        Changed(this, EventArgs.Empty);
    }
}
```

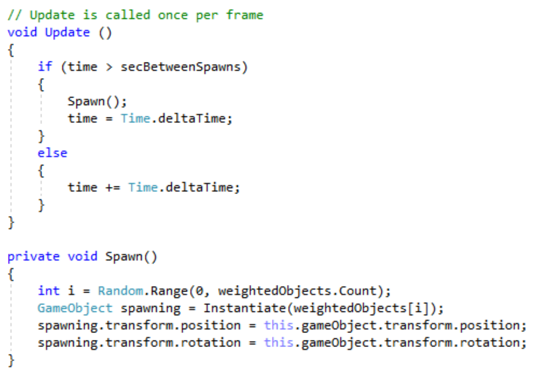
Figure 2: An excerpt from the method SeaGrid.AddShip() from this unit's BattleShips group project. The finally *clause of the* try-catch-finally *statement was throwing an exception, freezing the game. Without unit tests, the group members that worked to find and fix this problem, myself included, had to start at the GameController class and work through 6 to 8 layers of method calls and inheritance to find this section throwing exceptions. It took about two hours to find and fix this bug, time that could have been saved if the code we were working from had unit tests. And this isn't even the worst offender in this project; why the player is unable to shoot the AI's ships is still unresolved, and would probably have be far more easily addressed if unit testing had been implemented from the start (assuming that particular bug in the game's code is theoretically unit testable, of course; see below).*

- Code that is made more modular to conduct appropriate unit testing, or to pass said unit tests, becomes more reusable, as methods that are formed from code extracted out of larger methods can then be called by more than just the first method (for example, see fig. 2).
- Code that is made less interdependent to allow units to pass their unit tests will have the additional benefit of having less impact if a bug is introduced in such code.

That said, unit testing is not without its limitations, which include the following (SetSkill):

- It cannot guarantee the absence of errors, only that it did not find any errors.

- It can only be used to test the functionality of independent units; testing the program as a whole, or identifying performance and integration errors, are all beyond the scope of unit testing.
- There are limits to the number of scenarios and parameters and the amount of data that can be realistically used to test the source code. While it might be possible to test absolutely everything in this manner, it would be time consuming and impractical in many cases, and therefore not a viable option.



```
// Update is called once per frame
void Update ()
{
    if (time > secBetweenSpawns)
    {
        Spawn();
        time = Time.deltaTime;
    }
    else
    {
        time += Time.deltaTime;
    }
}

private void Spawn()
{
    int i = Random.Range(0, weightedObjects.Count);
    GameObject spawning = Instantiate(weightedObjects[i]);
    spawning.transform.position = this.gameObject.transform.position;
    spawning.transform.rotation = this.gameObject.transform.rotation;
}
```

Figure 3: An excerpt from the class SpawnItemScript *from my group project in* Digital Game Prototyping Lab. *The* Spawn() *method as shown is much more modular than if the code that comprised it were contained in* Update()*'s if statement. As such, the class can be updated later to call just* Spawn() *and access it's services in a more varied number of ways. Although it was not made this way to pass a unit test, it does demonstrate the benefit of increased modularity, which changing and refactoring code to pass unit tests encourages.*

Can I eat it now?

What does it taste good with?

While unit testing by itself is quite beneficial, it isn't the solution for identifying all bugs. As noted above, the scale of code that unit testing works with is that of individual, isolated units, meaning that a range of bugs are beyond its scope. Therefore, it should form one part of a larger testing scheme, supplementing and be supplemented by other types of testing, such as component testing (testing classes, packages, small programs, or other program elements), integration testing (assessing the combined execution of multiple classes, packages, components, or subsystems), and system testing (assessing the execution of the software product as a whole, including how it interacts with other software and hardware systems). Each covers the weaknesses of the others, and as such should be used in combination (as appropriate and necessary) to ensure the integrity of the software product being developed (Baruwal Chetri 2018).

Each of these types of testing, but especially unit testing, can also be substantially enhanced by a version control system. If later versions of a unit (or any larger division of code) fail a test it had previously passed, the version control system can provide a list of changes (if any) that have been made to the unit since it last passed (OneStopTesting.com a). As such, version control and (unit) testing go hand in hand.

What does it taste not-so-good with?

There is one type of program that doesn't mesh quite as well with unit testing as regular software: computer games. That's not to say unit testing can't be included in games at all; it can be useful for testing some parts of the game's code, such as (Guibert 2017):

- Maths functions (e.g. interpolation, combination, min/max)
- Generic helper functions, such as for string management, date/time management, or file compression.
- Code for the serialization/unserialization of game saves.

- Code for level generation, game hints, or the players' usage of in-game currencies or resources.
- Code involved in loading textures or other assets (Gilbert, 2018).

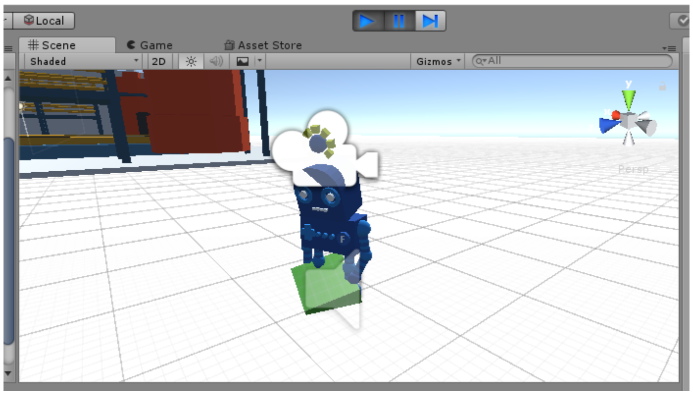


Figure 4: A screenshot of my group's game prototype in Digital Game Prototyping Lab*, where the player has picked up an object in their left hand, and that object is now cutting into both the player's avatar and the ground.*

Where it would be feasible to unit test a particular piece of a game's code, go nuts. However, while it is important that unit testable-code works as it should, unit testable-code is not where most bugs in games come from. The majority of bugs pop up when the player is doing something normal and the outputs for the involved functions are technically correct, but the game logic for that task is flawed (Gilbert 2018). There are other bugs where the game works fine, but something is visually wrong, such as objects overlapping with each other or the ground (fig. 4). Then there's other features that have to be tested for bugs manually, such as whether or not the game is detecting inputs properly (Guibert 2017).
These three kinds of bugs (and more besides) are hard to unit test realistically (Gilbert 2018, Guibert 2017). Their failure states are hard to identify, and the cost and effort involved quickly outweigh the benefits (Gilbert 2018). Instead, these bugs should be checked for through playtesting by both the developers and dedicated beta testers before the game is released. Though it might not be as precise as unit testing in terms of identifying what line of code is acting up, playtesting will in all likelihood identify problems that are within unit testing's scope. As such some might argue that playtesting makes unit testing a little bit redundant and therefore not worth the effort as, at the end of the day, unlike programs needed for MRIs or rockets, a small bug in a game that could have been picked up through unit testing but wasn't isn't going to kill anyone.

Ok, ok, how should I cook with it?

Though unit testing is not a remedy to cure all ills, there are bugs for which it is suited, and should be implemented to identify these, and implemented properly. Good practices to observe when employing unit testing include the following:

Clarity of test methods

- Name test methods to indicate what they test (Harper 2012b, Pylons Project).
- Readability of how a unit test works is more important than not duplicating functionality in tests. Having to change four or five similar tests is better than not understanding one non-duplicated test when it fails (Harper 2012c).

What to test

- Make each test method test just one thing (Harper 2012a, Pylons Project).
- Aim to cover all paths through the unit, paying particular attention to loop conditions (STF).
- Don't create test methods for everything; focus on the tests that impact the system's behaviour (STF).
- Write test methods not only to verify the behaviour of units, but their performance as well (STF).

- Don’t test the inner workings of a unit and how it was implemented. Unit tests should focus on the output generated instead; how that output is generated may change, and if the unit test is implementation-specific, the modified code will fail regardless of if it provides the correct output or not (Harper 2012c).

Test data

- Use mock objects (OneStopTesting.com b) and only add enough to those objects to make the dependent tests pass (Pylons Project).
- That said, use testing data that is as realistic as possible (STF).

Integrity of test methods

- Write test methods that are self-sufficient and independent of each other (Harper 2012b, STF).
- Keep the development and testing environments isolated (STF).
- Tests should be deterministic, either passing all the time, or failing until the unit being tested is fixed (Harper 2012b).

When doing unit testing

- Run tests continuously and frequently (STF).
- Make sure you have a sustainable process for reviewing test case failures daily and addressing them immediately. If such a process isn’t implemented, the software and unit tests won’t match up, creating false positives and reducing the effectiveness of the testing (OneStopTesting.com a).
- Before fixing a problem, make sure you’ve got a test that exposes it; write one if you don’t (STF). That way:
    - If you haven’t fixed it properly, this test will tell you.
    - Your unit testing suite becomes more comprehensive as a result.
    - You ensure that the test gets written; if you fix the bug first, you’ll probably be too lazy to write the unit test afterwards.
- Keep records of tests that have been performed and changes that have been made to the source code of any of the units. A version control system is essential here, since if later versions of a unit fail a test it had previously passed, the version control system can provide a list of changes (if any) that have been made to the unit since it last passed (OneStopTesting.com a).

References

OneStopTesting.com a, *Limitations of unit testing*, OneStopTesting.com, viewed 27 September 2018, <http://www.onestoptesting.com/unit-testing/limitations.asp>.

OneStopTesting.com b, *Six Rules of Unit Testing*, OneStopTesting.com, viewed 27 September 2018, <http://www.onestoptesting.com/unit-testing/rules.asp>.

Pylons Project, *Unit testing guidelines,* Pylons Project, viewed 27 September 2018, <https://pylonsproject.org/community-unit-testing-guidelines.html>.

Set Skill, *Unit Testing*, WordPress Foundation, viewed 26 September 2018, <http://setskill.com/software-testing/unit-testing >.

STF, *Unit Testing*, WordPress Foundation, viewed 26 September 2018, <http://softwaretestingfundamentals.com/unit-testing/>.

Baruwal Chhetri, M 2018, *Lecture 6: Validation, Verification, and Unit Testing*, SWE20001: Development Project 1 – Tools and Practices, Learning materials on Blackboard, Swinburne University of Technology, 4 September, viewed 28 September 2018.

Gilbert, R 2018, *Unit Testing Games*, GrumpyGamer.com, viewed 28 September 2018, <https://grumpygamer.com/unit_testing_games >.

Guibert, F 2017, *Unit testing in video games,* UBM TechWeb, viewed 28 September 2018, <http://www.gamasutra.com/blogs/FrancoisGuibert/20170612/299785/Unit_testing_in_video_games.php>.

Harper, D 2012a, *8 Principles of Better Unit Testing* (page 1 of 3), Converge360, viewed 28 September 2018, <https://esj.com/Articles/2012/09/24/Better-Unit-Testing.aspx?Page=1>.

Harper, D 2012b, *8 Principles of Better Unit Testing* (page 2 of 3), Converge360, viewed 28 September 2018, <https://esj.com/Articles/2012/09/24/Better-Unit-Testing.aspx?Page=2>.

Harper, D 2012c, *8 Principles of Better Unit Testing* (page 3 of 3), Converge360, viewed 28 September 2018, <https://esj.com/Articles/2012/09/24/Better-Unit-Testing.aspx?Page=3>.

Rouse, M, *unit testing*, TestTarget, viewed 26 September 2018, <https://searchsoftwarequality.techtarget.com/definition/unit-testing>.