COMP110: Principles of Computing

Software Testing

Today's lecture

Today's lecture has **three parts**

- Software testing and test-driven development
- Introducing COMP110 Coding Task II
- Object composition in C++

Software testing

In this section

In this section you will learn how to:

- **Discuss** the importance of software testing in game development
- **Identify** the different types and levels of testing
- **Apply** test-driven development practices to your own programming projects

Further reading

- Pressman, R.S. (2009) Software Engineering: A Practitioner's Approach. 7th Edition. McGraw-Hill.

Quality

Last time:

- There are many ways of measuring the **quality** of a game or piece of software
- **Quality assurance** is important to ensure that the software is of sufficiently high quality to provide benefit to developers and end users

Testing

- Finding **inadvertent errors** in the design and implementation of software
- Often takes more time and effort than any other part of development
- ... but letting errors slip into the final product can be even more costly
- Testing $\neq$ quality assurance
  - Testing is an important part of QA, but **not the only part**

Who is responsible?

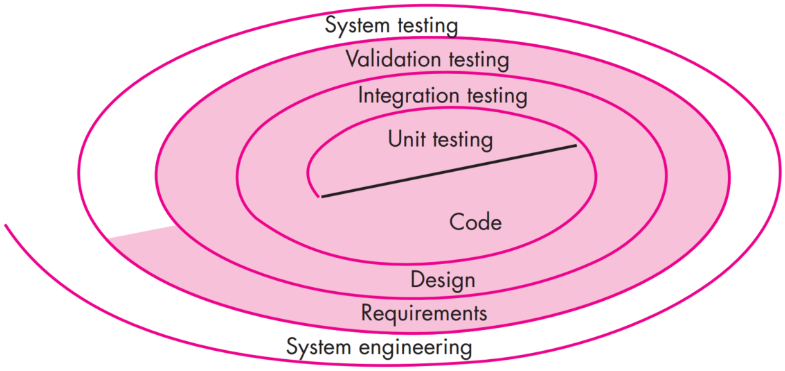
- Last time, we discussed that **designers**, **developers**, **publishers**, and maybe even **players** share the responsibility for software quality in games
- Who should take responsibility for **testing**?
  - "**Developers** write the code, so they should make sure it works"?
  - "**Everyone** is responsible for quality, so everyone should pitch in"?
  - "Code should be tested by **someone other** than the developer who wrote it"?

Socrative `6E8NSW3IN`

So who should test game software?

- In pairs.
- Discuss for 2-minutes.
- **Suggest** which parties should take responsibility for testing **and justify** your answer.

Testing strategy



(Pressman, 2009) Figure 17.1

Testing strategy

- Development starts with system engineering and works **inwards**
  - The **waterfall model**
  - Agile doesn't quite work like this
- Testing starts with unit testing and works **outwards**
- **White box testing**: testing the software **with** knowledge of its internal workings
- **Black box testing**: testing the software **without** knowledge of its internal workings

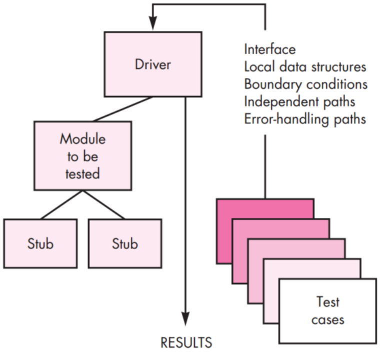
Unit testing

- A **unit test** is a piece of code that verifies a **unit** (e.g. a function or class) of a program
- E.g. verifies that a function called with a particular set of parameters returns the expected result
- E.g. verifies that a function called with invalid parameters throws the expected error

Designing user tests

- Test the **edge cases**
  - Programming errors often occur at the **boundary** between valid and invalid input, or the boundary between one case and another
  - E.g. for an n-element data structure, test accessing elements $n - 1$, n, $n + 1$
- Aim for high **coverage**
  - Ideally, **every line of code** should be executed in **at least one** unit test

Drivers and stubs



(Pressman, 2009) Figure 17.4

- Unit testing generally requires extra code to be written
- **Driver** — to set up any required state and run the test
- **Stubs** — to replace any modules upon which the module under test depends

Integration testing

- Verify that the individual units work **together**
- Can be done **top-down** or **bottom-up**
- Either way, the idea is to gradually replace stubs and drivers with actual units, testing as you go
- **Regression testing** is important — re-running tests to ensure that recent additions have not broken anything

Socrative `6E8NSW3IN`

If the units have been thoroughly tested individually, why is integration testing needed?

- In pairs.
- Discuss for 2-minutes.
- Give an **example** of a problem that integration testing might uncover, but that unit testing might miss.

Validation testing

- Testing the complete software system from the user's point of view
- E.g. playtesting

Socrative `6E8NSW3IN`

If unit testing and integration testing have been done correctly, why is validation testing needed?

- In pairs.
- Discuss for 2-minutes.
- Give an **example** of a problem that validation testing might uncover, but that unit and integration testing might miss.

When is testing “done”?

- The aim of testing is to find bugs, so it’s done when there are no bugs left to be found! ☺
- When the software is (quantitatively or qualitatively) “good enough”
- Testing is never “done” — the burden just shifts onto the users

Test driven development (TDD)

- A development process that advocates writing the unit tests **first**
- Repeat the following three steps:
  1. **Red**: create a new test case, which should initially **fail**
  2. **Green**: write code to make the new test **succeed** (without causing the other test cases to fail)
  3. **Refactor**: **improve** the code, ensuring that all tests still **succeed**

Why TDD?

- Often easier to convert a **user story** into test cases rather than directly into code
- Writing the bare minimum of code to make the test "green" lets you **focus on user stories**, not on **over-generalisation** or **non-essential functionality**
  - **KISS**: Keep It Simple, Stupid
  - **YAGNI**: You Aren't Gonna Need It

Red

- Create a new test case, which should initially **fail**
- Write only enough code to allow the test case to compile and run, e.g. write a **stub** function
- What if the test succeeds?
    - Maybe you already implemented that feature?
    - Maybe the test case is wrong?
    - Maybe your unit testing code is broken?

Green

- Add the **bare minimum** of code to make the new test case succeed
  - **K**eep **I**t **S**imple, **S**tupid!
- Verify that **all** unit tests now succeed
- What if old tests now fail?
  - Fix it
  - **Or** revert and start again — can be faster than debugging
  - (you **did** commit before you started, right?)

Refactor

- E.g. remove duplication, improve names, add documentation, apply design patterns, ...
- To generalise or not to generalise?
- **Do** generalise if it makes the code **simpler**
- **Don't** generalise because you "might" need it later
  - **Y**ou **A**ren't **G**onna **N**eed **I**t!
  - Wait until it **is** needed in another cycle
- Verify that **all** unit tests still succeed

Socrative `6E8NSW3IN`

How suitable is the test driven approach for game development?

- In pairs.
- Discuss for 2-minutes.
- Suggest **one advantage and one disadvantage** of test driven development in the context of game development

Summary

- **Testing** is an important part of software quality assurance (but not the only part)
- There are several different **levels** of testing, which mirror the different levels of software development
    - Unit testing $\leftrightarrow$ Coding
    - Integration testing $\leftrightarrow$ Design
    - Validation testing $\leftrightarrow$ Requirement planning
- **Test driven development** is one possible strategy for testing your software (but not the only strategy)

COMP110 Coding Task 2

The assignment brief

LearningSpace: COMP110 assignment 4

The task

- Develop a **component**...
  - **For example**, non-player character AI
  - **or** procedural content generator
  - **or** physics simulation
  - **or** combat mechanic
  - **or** ...
- ... for a **game**
  - BA Digital Games project
  - **or** your COMP150 group project
  - **or** your COMP130 Kivy project

How does this fit with COMP150?

- You will take **ownership** of this component of the game
    - Essentially as a “consultant” to your own team
- Members of the same COMP150 team **must not** target the same component of their COMP150 game

Proposal

- For **next Wednesday's COMP110 lecture (9th March)**
- See assignment brief for details

Composition in C++

From COMP110 session 7

OOP models three types of relationship:

- **Is-a**: modelled by **instantiation**
- **Has-a**: modelled by **composition**
- **Is-a-type-of**: modelled by **inheritance**

Composition in Python

- "A duck has a bill" $\rightarrow$ "Each instance of class Duck contains a reference to an instance of class Bill"

```
class Bill:
    ...

class Duck:
    def __init__(self):
        self.bill = Bill()
```

- Why a **reference**?
- Because that's your only option in Python!

Composition in C++

"A duck has a bill"

- "Each instance of class Duck contains **an instance** of class Bill"

```
class Bill { ... };

class Duck
{
private:
    Bill bill;
};
```

- **Or** "Each instance of class Duck contains **a pointer** to an instance of class Bill"

```
class Bill { ... };

class Duck
{
private:
    Bill* bill;
};
```

Composition in C++

- The contained instance of `Bill` is stored **inside** the instance of `Duck` (literally, in memory)
- It is constructed when the `Duck` instance is constructed, and destroyed when it is destroyed

- The contained instance of `Bill` is stored **outside** the instance of `Duck`, which only stores a **pointer**
- It is usually constructed manually using **`new`**, and so must be destroyed manually using **`delete`**

When to use each?

- Pointers are more versatile
  - Allow several pointers to the same instance (e.g. several ducks might **have-a** single pond)
  - Allow **circular references** (e.g. a duck **has-a** bill, and a bill **has-a** duck)
  - Pointers allow **polymorphism** (e.g. a pointer to a "duck" might actually be a pointer to a mallard)
- **But** stored instances are easier to work with
  - Destruction is handled automatically
- They model slightly different types of **has-a** relationship
  - Instance: **has-a** in the sense of "contains"
  - Pointer: **has-a** in the sense of "is associated with"

Circular references

- The following code won't compile:

```
class Bill
{
private:
    Duck* owner;   // Error here
};

class Duck
{
private:
    Bill bill;
};
```

What's the problem?

- **Order** of definitions and declarations matters in C++
- You **can't** use something **before** it's been declared
- The offending line is using `Duck` before it's declared
- Does this make circular referencing impossible? Need to declare `Duck` before `Bill`, but also need to declare `Bill` before `Duck`
    - No...

Forward declarations

- Solution: use a **forward declaration**

```
class Duck;    // Forward declaration

class Bill
{
private:
    Duck* owner;    // This is OK now
};

class Duck
{
private:
    Bill bill;
};
```

Socrative 6E8NSW3IN

- Different code, same problem:

Bill.h

```
#pragma once

#include "Duck.h"

class Bill
{
private:
    Duck* owner;
};
```

Duck.h

```
#pragma once

#include "Bill.h"

class Duck
{
private:
    Bill bill;
};
```

- How to fix it?
- Discuss **in pairs** for 2 minutes and post your answer

Limitations of forward declarations

- Basically all you can do with a forward declared class is declare a **pointer** to it
- E.g. this wouldn't work:

```
class Bill;

class Duck
{
private:
    Bill bill; // Error: undefined class 'Bill'
};

class Bill
{
private:
    Duck* owner;
};
```

Limitations of forward declarations

- The compiler needs to know **how big** (in bytes) an instance of `Bill` is, which the forward declaration doesn't tell it
- **All pointers have the same size**, so a forward declaration is enough in that case
- Circular references of contained instances are **impossible**
  - At least one of the links in the chain must be a **pointer**
  - "Contains-a" relationships in real life can't be circular either
  - Philosophical thought for the day: how big would something have to be, to be big enough to contain itself?

Composition and containers

```
std::vector<Duck> ducks;
```

- The instances are stored **consecutively** in memory
- What happens when the size of the `vector` changes?
  - **Recall**: when the size of a `vector` changes, a new array is allocated, the contents are **copied** into it and the old array is **destroyed**
- This can result in unexpected calls to your **copy constructor** and **destructor**
- Can cause problems when using certain idioms (e.g. **RAII**)

Composition and containers

```
std::vector<Duck*> ducks;
```

- This is a `vector` of **pointers**
- When the `vector` changes size, the instances stay where they are — only the **pointers** are copied
- **However**, managing instances with **`new`** and **`delete`** is now your responsibility

Ownership

- It is important to keep track of which module "**owns**" a particular instance
- The owner is responsible for `delete`ing the instance when it is no longer needed
- Code should **never** `delete` an instance that it does not own
- Generally ownership stays with the module that created the instance, **unless** it explicitly transfers it
    - In which case, **document this clearly** in the module documentation
    - If you take ownership of a pointer, `delete`ing it is now your responsibility
- NB: C++ doesn't care about ownership — it's a concept **we** use to write and understand programs

Summary

- **Composition** models **has-a** relationships, which can include **contains-a** and **is-associated-with-a**
- **Circular references** can be set up using pointers, but **forward declarations** are often needed to make the compiler understand them
- **Ownership** is one way of keeping track of instances and understanding when to `delete` them