Project Software Engineering
Unit Testing, Contracten

Kasper Engelen
kasper.engelen@uantwerpen.be

In deze sessie gaan we leren werken met twee zaken die aan bod gaan komen in het project: *software testing* en *design by contract*. Om te kunnen slagen voor het project is het essentieel dat je uitgebreid gebruikmaakt van deze twee tools.

Software testing wordt gebruikt om fouten te vinden in een softwaresysteem. Concreet houdt dit in dat je enkele (en soms zelfs honderden) scenario's uitwerkt en dan gaat kijken hoe je softwaresysteem zich gedraagt in zulke scenario's. Een testing *framework* laat toe om zulke testen te automatiseren en voorziet asserts om te controleren of de code doet wat die hoort te doen. Meer informatie kan je vinden in Appendix A.

Design by contract is een andere belangrijke techniek waarbij men enerzijds gaat controleren of een functie correct wordt opgeroepen (precondities) en anderzijds gaat men na of de functie doet wat die moet doen (postcondities). Design by contract kan gebruikt worden in combinatie met testen. Meer informatie kan je vinden in Appendix B.

Oefening 1: Installeren van Google Test (gtest)

Vooraleer we kunnen beginnen met het schrijven van tests, zullen we eerst een testing framework moeten installeren. Voor dit vak maken we gebruik van Google Test, ook wel "gtest" genaamd. Dit framework, geschreven in C++, laat toe om testen te schrijven, ze uit te voeren, en voorziet verschillende soorten asserts.

Concreet zal je eerst de instructies op de cursuswebsite moeten volgen voor de installatie. **Deze instructies zijn verschillend voor Windows enerzijds en MacOS/Ubuntu anderzijds.** Op de cursuswebsite vindt je ook het "TicTacToe" voorbeeld, dat ook gebruikmaakt van Google Test. Op deze website vind je extra informatie over de functionaliteit van het gtest framework.

Oefening 2: Het testen van een eenvoudige functie

Lees eerst de uitleg in Appendix A over Unit Testing. We zullen nu een aantal testen schrijven voor een functie `largest` die het grootste element uit een lijst returned. Beschouw onderstaande code:

```
int largest(vector<int> list){
    /* code */
}
```

We zien dat deze functie geen implementatie heeft. Desondanks weten we wat de functie doet, en welke input de functie nodig heeft. Deze twee dingen zijn genoeg om nuttige testen te bedenken te voor een functie. Schrijf minstens 5 testen in het Google Test framework voor deze functie. Denk aan Algemene correctheid, Randgevallen en Fouten.

Beschouw onderstaande (foute) implementatie van bovenstaande functie. Hoeveel van de testen die je hiervoor hebt geschreven falen hierop? In Appendix C kan je een implementatie vinden die beter werkt. Je vindt deze ook in de file `Largest.cpp`.

```
int largest(vector<int> list){
    int max = numeric_limits<int>::max();
    for(int i = 0; i < list.size() - 1; i++){
        if(list[i] < max){
            max = list[i];
        }
    }
    return max;
```

```
}
```

Oefening 3: Het testen van een klasse

Om wat meer uitdaging te hebben, gaan we nu proberen om testen te schrijven voor een klasse. In principe is dit net hetzelfde als een test schrijven voor een functie, maar dan met methodes. Het enige verschil is dat methodes worden opgeroepen op een klasse en dat een klasse variabelen bevat die het gedrag van de methode beïnvloeden. Je zal dus rekening moeten houden met de argumenten die je meegeeft aan de constructor, in welke volgorde je de methodes oproept, etc.

In de file `Fighterplane.cpp` vindt je een klasse `Fighterplane`. De bedoeling is om testen te schrijven voor deze klasse. Om alles ordelijk te houden, schrijf je de testen best in een aparte file (die je dan ook zal moeten vermelden in je `CMakeLists.txt`).

Oefening 4: Design by contract

Lees de uitleg in Appendix B over Contracten. In deze oefening gaan we enkele contracten, t.t.z. pre- en postcondities, schrijven voor de code van de vorige twee oefeningen. Maak steeds gebruik van de `REQUIRE` en `ENSURE` macro's in de header file `DesignByContract.h`. Merk op dat je deze ook moet gebruiken in het project.

Oefeningen:

1. Beschouw de functie `largest` uit appendix C (en in `Largest.cpp`). Voor inspiratie voor de contracten kan je kijken naar de tests die je voor deze functie hebt geschreven.

2. Beschouw de klasse `Fighterplane` in de file `Fighterplane.cpp`. Schrijf enkele pre- en postcondities voor de methodes van deze klasse. Je kan opnieuw inspiratie halen uit de testen voor deze klasse.

3. Nadat je de contracten hebt opgesteld, zal je enkele testen moeten schrijven om te verifiëren dat deze contracten correct werken.

    - Stel een happy day scenario op waarbij er aan de contracten is voldaan.

    - Stel ook enkele tests op waarbij er niet aan de contracten is voldaan. Je kan dit doen d.m.v. `EXPECT_DEATH`.

Appendix A: Unit Testing

Software Testing heeft als doel het evalueren van een bepaalde eigenschap of functionaliteit van een programma of een systeem, en te verifiëren dat dit overeenkomt met het verwachtte resultaat. Er zijn verschillende soorten van Software Testing:

- **Unit Testing:** Verifieert de functionaliteit van een specifiek stuk code, meestal één specifieke klasse, of één specifieke functie.
- **Integration Testing:** Het testen van enkele modules die moeten samenwerken. Dit is nuttig want soms kan het zijn dat een systeem als geheel faalt, terwijl de afzonderlijke onderdelen wel correct werken.
- **System Testing:** Testen van het volledig systeem.
- **System Integration Testing:** Testen van de samenwerking tussen verschillende systemen.

In dit project zullen we vooral focussen op unit testing en integration testing. Merk ook op dat je meerdere tests kan schrijven voor éénzelfde functie of klasse. Daarnaast is het bij testing ook belangrijk om zo granulair mogelijk te werken: maak elke test zo klein en zo specifiek mogelijk.

Een goede Unit Test is:

- **Herhaalbaar:** Elke keer dat de test wordt uitgevoerd, moet deze hetzelfde resultaat bekomen. (vermijd random generators, code die afhangt van de "current time", etc.)
- **Onafhankelijk:** Test één methode of functie per unittest. Verschillende testen mogen niet afhankelijk zijn van elkaar.
- **Waardevol:** Zorg dat testen nuttige dingen verifiëren en de klasse of functie onderwerpen aan interessante scenario's. Testen dat "1 + 1 = 2" is dus niet echt nuttig. Je kan hierbij steunen op de use cases: "wat moet de use case doen?", "onder welke omstandigheden vindt de use case plaats?"
- **Grondig:** Test ALLE pre/post condities, randgevallen, exceptions, etc.

Om grondig te testen, moet je een aantal dingen nakijken:

- **Algemene correctheid:** Dit zijn de zogenaamde "Happy Day" testen. Deze controleren dat het systeem, onder normale omstandigheden, doet wat het moet doen.
- **Randgevallen:**
  - *Orde:* Heeft een andere volgorde een effect op het resultaat?
  - *Bereik:* Hoe reageert het component op het getal nul, het minimum, het maximum, positieve waarden, negatieve waarden, etc.
  - *Bestaan:* Wat als je een `nullptr` meegeeft als parameter? Wat met lege verzamelingen (lege array, lege list, lege vector, etc.)?, wat met lege Strings?
  - *Kardinaliteit:* Wat is het verwachtte aantal items?
- **Fouten:** Worden de juiste foutboodschappen gegeven? Wat met I/O issues, zoals ontbrekende bestanden, onleesbare of lege bestanden? Je kan hier proberen om opzettelijk de functie te doen falen, zodat je kan verifiëren dat het systeem correct om kan gaan met fouten.

Appendix B: Contracten

Contracten worden gebruikt om jezelf ervan te verzekeren dat je software juist is opgebouwd. Het is een manier om formeel te documenteren welke functionaliteit een bepaalde functie of methode heeft. Je kan dit programmatisch vast leggen met behulp van asserts (of in ons geval met de macro's `REQUIRE` en `ENSURE`, gedefinieerd in DesignByContract.h).

We gebruiken `REQUIRE` voor precondities en `ENSURE` voor postcondities. Met `REQUIRE` kan je zaken "eisen" van degene die de functie oproept. Je kan bijvoorbeeld eisen dat een parameter `x` groter moet zijn dan 5. Met `ENSURE` leg je vast wat de functie die je hebt geschreven moet doen. `ENSURE` specificeert dus in welke staat het systeem zich bevindt nadat de functie is uitgevoerd.

Het is de taak van de code die de functie of methode oproept om ervoor te zorgen dat het systeem zich in de juiste staat (zoals beschreven in de preconditie) bevindt. Je moet dus in de functie of methode zelf geen extra controle's meer implementeren en foutboodschappen genereren, wanneer het vanuit de verkeerde toestand wordt opgeroepen. Deze controles en foutboodschappen moeten dus in de oproepende kant worden geïmplementeerd.

Elke kant van een methode/functie oproep haalt voordelen uit een contract, maar moet wel zijn verplichtingen nakomen:

Contract	**Voordelen**	**Verplichtingen**
Methode of functie met pre- en postcondities (de PROVIDER)	• Geen nood om de input waarden na te kijken. • De input voldoet gegarandeerd aan de preconditie.	Moet er voor zorgen dat er aan de postconditie voldaan is.
Code die de functie oproept (de CLIENT)	• Geen nood om de output waarden na te kijken. • Het resultaat voldoet gegarandeerd aan de postconditie.	Moet er voor zorgen dat aan de precondities van de Provider voldaan wordt.

Appendix C: Een betere implementatie voor de functie largest

```
#include <exception>
#include <vector>

using namespace std;
class IllegalArgException: public exception{
    const char* message;
public:
   IllegalArgException(const char * message){
        this->message = message;
    }
   virtual const char* what() const throw(){
        return message;
   }
};


int largest2(vector<int>* list) {
    if (list == nullptr) {
        throw new IllegalArgException("list cannot be nullptr");
    } else if (list->empty()) {
        throw new IllegalArgException("list cannot be empty");
    }
    int max = numeric_limits<int>::min();
    for (int i = 0; i < list->size(); i++) {
        if (list->at(i) > max) {
            max = list->at(i);
        }
    }
    return max;
}
```