Duale Hochschule Baden-Württemberg

Advanced Software Engineering 2

–

Dokumentation

PIC-Simulator

David Eymann, Tom Wagner

Dozent
Daniel Lindner

30. September 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Projektbeschreibung

«««< Updated upstream Als Basis für diese Arbeit dient ein PIC-Simulator, PIC steht hierbei für ein Mikrocontroller von Microchip Technology[1]. Der Simulator ist in C# geschrieben und mit Windows Forms Erhält er seine Grafische Oberfläche. **Das für diese Abgabe relevante Repository befindet sich unter:**
`https://github.com/tomwgnr/ASE-PIC_Simulator`
======= Als Basis für diese Arbeit dient ein PIC-Simulator, PIC steht hierbei für ein Mikrocontroller von Mikrochip Technology[2] . Der Simulator ist in C-# geschrieben und mit Windows Forms Erhält er seine Grafische Oberfläche. **Das für diese Abgabe relevante Repository befindet sich unter:** `https://github.com/tomwgnr/ASE-PIC_Simulator` »»»> Stashed changes

1.1 Einrichtung

[1] `https://en.wikipedia.org/wiki/Microchip_Technology`
[2] `https://en.wikipedia.org/wiki/Microchip_Technology`

Kapitel 2

Entwicklung

2.1 Clean Architecture

2.2 Refactoring

2.2.1 Switch-Case in der CPU

Natürlich musste der 220 Zeilen lange Switch-Case welcher in der `CPU` Klasse die Befehle aufgrund des eingegebenen `OpCode` gewählt und die hierfür nötigen Berechnungen direkt im Speicher ausführte. Dieser wurde in die Klassen `Memory` und `Command` unterteilt. Die Auswahl wird jetzt im `Decoder` erledigt. Da die einzelnen Befehle sich sehr ähneln dient die Klasse `Command` nur als Vorlage für die Klassen der Befehle im Namespace commands. Hier werden Befehlsspezifische Aufgaben erledigt wie die Ausführung des jeweiligen Befehls aber auch die Überprüfung des OpCodes, um den Decoder sauberer zu halten.

2.2.2 Register

Da die Register auch mit vielem Anderen noch in der `Simulator` Klasse waren wurden diese auch in Einzelne Klassen im Namespace `register` untergebracht. Wie auch bei den Befehlen sind hier grundlegende Aufgaben wie das erstellen des Registers nun in jeder Klasse für jedes Register als Methode untergebracht.

2.2.3 Interupt- und FlankenCheck

Auch die Methoden für die Checks auf Interrupts und Flanken wurden im Rahmen des Refactoring in eigene Klassen gepackt um die Simulator Klasse zu vereinfachen.

2.3 Unit-Tests

Die Unit-Test können im Ordner /PicSimTest gefunden werden. Mit diesen Tests werden viele unserer Klassen getestet. Um die (Unit-)Tests für C# zu schreiben wird hier das MSTest Framework genutzt welches es in Kombination mit Visual Studio ermöglicht die Tests automatisch laufen zu lassen.

2.3.1 Einsatz von Mocks

Zum Testen einiger Methoden werden Mock-Objekte erstellt um die Test so abgetrennt wie möglich halten zu können. Um dies einfach machen zu können wird moq[1] genutzt. Moq ermöglicht es einfach aus Klassen Mock-Objekte zu erstellen welche dann zum Testen genutzt werden können.

2.3.2 ATRIP-Regeln

Automatic

Visual Studio ermöglicht es mit einem Klick alle Test Durchlaufen zu lassen und findet hier auch neue Test im Projekt `PicSimTest` wenn diese mit den richtigen Attributen `[TestClass]` oder `[TestMethode]` versehen wurden.

Thorough

Da die Klassen der Befehle und der Register den Großteil der Rechenarbeit des simulierten Pic erledigen werden diese auf ihr verhalten getestet.

Repeatable

Independent

Um Unabhängigkeit voneinander zu haben sind die Tests für eine Klasse jeweils in einer eigenen TestClass und darin sind nur die Testmethoden für die Methoden der zu Testenden Klasse enthalten. Durch die Konzeption der Tests können diese in belibiger Reihenfolge oder auch einzeln Ausgeführt werden. Dies Ermöglicht es auch einzelne Programmabschnitte zu testen ohne auf viele Tests warten zu müssen.

Professional

Um die Absicht des Tests klar und und einfach verständlich zu gestalten sind die Testmethoden und Testklassen Namen selbsterklärend gewählt. So heißt die Klasse des Test für

[1] `https://github.com/moq/moq4`

den im Simulator genutzten Befehl `Addlw` `AddlwTest`. Auch die Test Methoden selbsterklärend so wird die Methode `isOpCode` im Objekt Addlw von der Methode `isOpCode_rightCode` auf Verhalten bei Eingabe des richtigen Codes getestet.

2.4 Programming Principles

2.4.1 SOLID

Single responsibility principle

Open/Closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

2.4.2 GRASP

High Cohesion

Low Coupling

2.4.3 DRY – Don't Repeat Yourself