Dokumentation Programmentwurf PIC16F84-Simulator

Einleitung

Ziel dieser Dokumentation für die Vorlesung Advanced Software Engineering ist die Zusammenfassung aller unternommenen Tätigkeiten, um den Quellcode eines im Rahmen der Vorlesung Rechnertechnik entwickelten PIC16F84-Simulators in „guten“ Code umzuwandeln.

„Guter Code“ setzt sich dabei aus mehreren Aspekten zusammen. Abgedeckt werden sowohl die grundlegenden Funktionskonzepte des Programms als auch die Darstellung des Codes und die Anpassbarkeit bei eventuell notwendigen Änderungen in der Zukunft.

Da dieser Programmentwurf auf einem bereits bestehenden Projekt fußt, wird das vor allem das Umsetzen von architekturellen und strukturellen Programmier-Regeln schwierig bis nahezu unmöglich. Um das Projekt nicht von Grund auf neu entwickeln zu müssen, werden diese Aspekte des Software Engineerings lokal begrenzt eingearbeitet. Da sie allerdings global skalieren, ist mit entsprechendem Mehraufwand die Umstrukturierung des gesamten Projekts dennoch möglich. Aus Gründen der Verständlichkeit wird darauf aber verzichtet, und stattdessen ein Proof of Concept beziehungsweise ein Proof of Understanding ausgearbeitet. Analog zu einem Induktionsbeweis besteht nicht die Notwendigkeit einer Brute Force ähnlichen Abarbeitung der Regelungen für jeden Anwendungsfall; stattdessen genügt der Beweis der Anwendbarkeit auf einen Einzelfall und der Nachweis, das Konzept auf größere Maßstäbe zu skalieren.

Zur Versionskontrolle wird Github benutzt, das Repository ist unter folgendem Link zu finden:
https://github.com/Fireworker2000/softweng
Um den Simulator zu starten muss die SimGui-Klasse ausgeführt werden. In ihr befindet sich auch die Main-Methode.

Wichtig zu beachten bei diesem Projekt ist, dass der Code dieses Simulators stellenweise von den Empfehlungen beziehungsweise Vorgaben zu gutem Programmcode abweicht. Diese Abweichungen sind teilweise darauf zurückzuführen, dass bei der Entwicklung des Simulators versucht wurde, möglichst nah an der Dokumentation des Microchips (PIC16F84) zu bleiben. Die zum Teil erschwerte Lesbarkeit des reinen Codes wird also dadurch wettgemacht, dass er unter Zuhilfenahme der Dokumentation weitaus einfacher zu verstehen ist.

Da die Dokumentation des Microcontrollers wie bereits beschrieben essenziell zum Verständnis des Programmcodes ist, ist sie ebenfalls im Projektordner auf Github enthalten. Welche Seiten der Dokumentation relevant sind, wird an entsprechenden Stellen im Code und diesem Dokument auch nochmal erwähnt.

Programming Principles

In der Software-Entwicklung gibt es verschiedene Prinzipien, nach denen gute Programmierer ihren Code strukturieren. Unter anderem sind dies die SOLID-Prinzipien nach Robert C. Martin, wobei SOLID ein Akronym ist aus:

- Single Responsibility Principle:
  eine Zuständigkeit pro Klasse; nur ein Grund, sie zu ändern
- Open/Closed Principle:
  Erweiterung einer Klasse möglich, aber ohne Notwendigkeit einer Modifikation
- Liskov Substitution Principle:
  Objekte der Oberklassen problemlos austauschbar durch die der Unterklassen
- Interface Segregation Principle:
  Interface-Implementationen nicht zur Übernahme unnötiger Funktionalität zwingen
- Dependency Inversion Principle:
  Klassen-Abhängigkeit von wichtig zu unwichtig und von Abstraktion zu Konkretisierung

Zur Erfüllung des Single Responsibility Anspruchs wurde die Klasse Processor aufgeteilt. Die in dem Zuge neu geschaffene Klasse Decoder ist dafür zuständig, die eingelesene Codezeile aufzuteilen in den auszuführenden Befehl und dessen Operanden/Parameter. Die Codezeile ist als 14-bit Zahl hinterlegt, deshalb wird sie dem Decoder als Integer übergeben. Die konkrete Dekodierung wird abhängig vom Precommand ausgestaltet. Dieser besteht aus den ersten 2 Bits des Opcode und bestimmt die Bedeutung der folgenden Bits. Dazu Seite 56 des Microcontroller-Handbuchs:

PIC16F8X

TABLE 9-2 PIC16FXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb	LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS							
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff		C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff		Z	1,2
CLRF	f	Clear f	1	00 0001 1fff ffff		Z	2
CLRW	-	Clear W	1	00 0001 0xxx xxxx		Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff		Z	1,2
DECF	f, d	Decrement f	1	00 0011 dfff ffff		Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff			1,2,3
INCF	f, d	Increment f	1	00 1010 dfff ffff		Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff			1,2,3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff		Z	1,2
MOVF	f, d	Move f	1	00 1000 dfff ffff		Z	1,2
MOVWF	f	Move W to f	1	00 0000 1fff ffff			
NOP	-	No Operation	1	00 0000 0xx0 0000			
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff		C	1,2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff		C	1,2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff		C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff			1,2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff		Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS							
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff			1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff			1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff			3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff			3
LITERAL AND CONTROL OPERATIONS							
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk		C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk		Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk			
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100		$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk			
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk		Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk			
RETFIE	-	Return from interrupt	2	00 0000 0000 1001			
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk			
RETURN	-	Return from Subroutine	2	00 0000 0000 1000			
SLEEP	-	Go into standby mode	1	00 0000 0110 0011		$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk		C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk		Z	

O – Open/Closed Principle

Auch die nahezu aufwandslose hohe Anpassbarkeit nach dem Open/Closed Principle ist gewährleistet. Als Beispiel dafür kann die Einführung des Bridge Patterns betrachtet werden. Durch die Trennung von Decoder- und Logger-Implementation können beide problemlos erweitert werden ohne eine Modifikation an der jeweils anderen Klasse zu benötigen. Mehr Informationen zu der Funktionsweise des Bridge Patterns befinden sich im Kapitel Entwurfsmuster.

L – Liskov Substitution Principle

Damit dieses Programmierprinzip eingehalten wird, sollte im Idealfall jede Instanz einer Superklasse durch die einer Subklasse ausgetauscht werden können ohne unerwartetes Verhalten oder gar Probleme zu produzieren.

Im ursprünglichen Simulator-Quellcode existieren keine Vererbungsbeziehungen, daher ist das Liskov-Substitution Principle dort auf die einfachste mögliche Weise erfüllt. Beim erstellen des Bridge Patterns wurden allerdings Vererbungsstrukturen eingeführt. Jedoch ist die Klasse AbstractDecoder als abstrakte Klasse ausgelegt und es können somit keine Instanzen von ihr erstellt werden. Unter der Annahme, dass dies dennoch möglich wäre, wird beim Ersetzen eines abstrakten Decoders durch einen normalen Decoder das LSP nicht verletzt. Es wird lediglich Funktionalität hinzugefügt (in Form der Implementierung der decode-Funktion des Interface), es werden weder Methoden überschrieben noch Implementation ersetzt.

I – Interface Segregation Principle

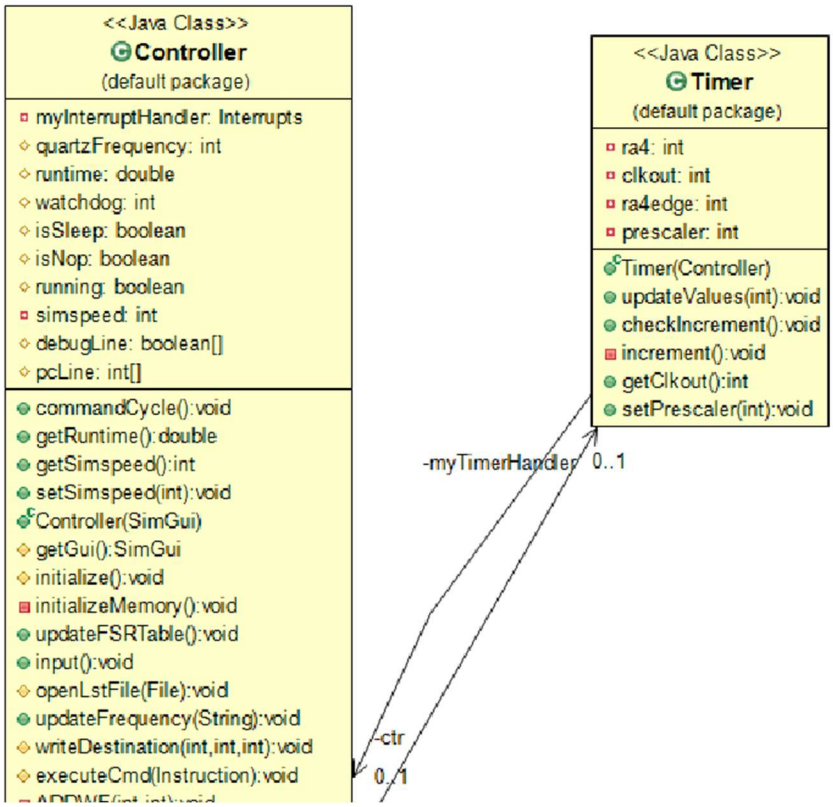
Für die Decoder-Klasse wurde auch ein Interface erzeugt, über die der Prozessor mit ihr kommunizieren kann. Damit das Interface Segregation Principle erfüllt ist, wurde im Interface nur die notwendigste Funktionalität des Decoders festgelegt:

```
public interface decoderInterface {

    public Instruction decodeCodeline(int codeline);

}
```

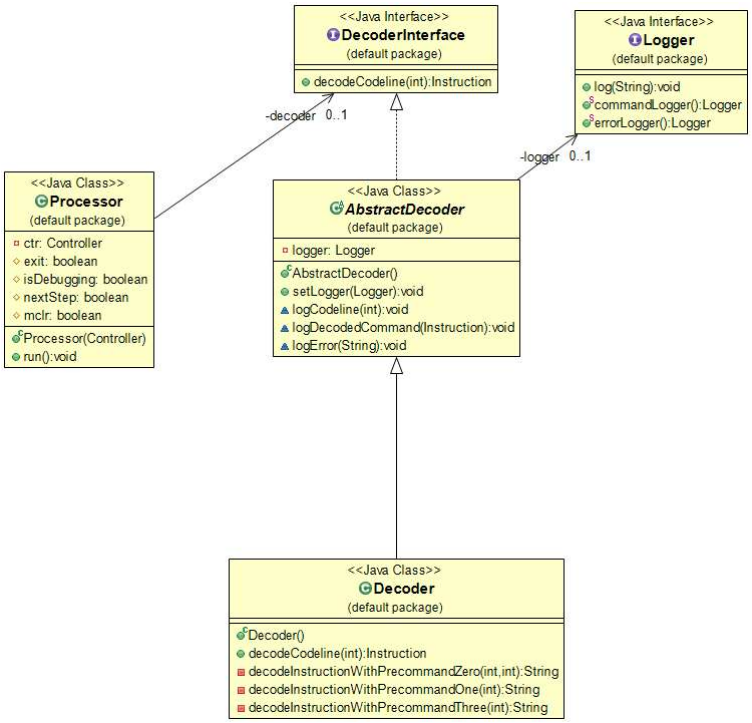
Obwohl der Decoder weitere Funktionen hat, um die übergebene Programmzeile in Abhängigkeit der Precommands aufzulösen, wird beim Interface nur die zentrale Funktion der Decodierung vorgegeben. Auf diese Weise bleibt es möglichen Variationen zukünftiger Decoder-Implementierungen frei, wie sie die Programmzeile aufschlüsseln. Konkret bedeutet dies, dass eine Entschlüsselung auf Basis des Precommands lediglich durch den Decoder selbst festgelegt wird, das Interface bleibt dabei außen vor. Sollte in Zukunft die Decodier-Funktionalität abgeändert werden müssen, kann der Code der Processor-Klasse nahezu gleich behalten werden, nur auf Seiten des Decoders müssen die entsprechenden Veränderungen vorgenommen werden.

D – Dependency Inversion Principle

Bei Betrachtung des UML-Diagrams fällt auf, dass jede Klassenbeziehung bidirektional implementiert wurde, hier beispielhaft dargestellt durch die Beziehung zwischen Timer und Controller:



Dies stellt eine klare Verletzung des LSP dar, da so beide Klassen gegenseitig voneinander abhängig sind. Entweder ist also jede Funktionalität auf gleicher Ebene implementiert oder die Regeln der Clean Architecture wurden missachtet. Bei Einführung der neuen Klassen jedoch wurde darauf geachtet, das D der SOLID-Prinzipien zu erfüllen. Im untenstehenden UML-Diagramm ist zu erkennen, dass Abhängigkeiten nur von unten nach oben bestehen, nicht jedoch anders herum:



Clean Architecture

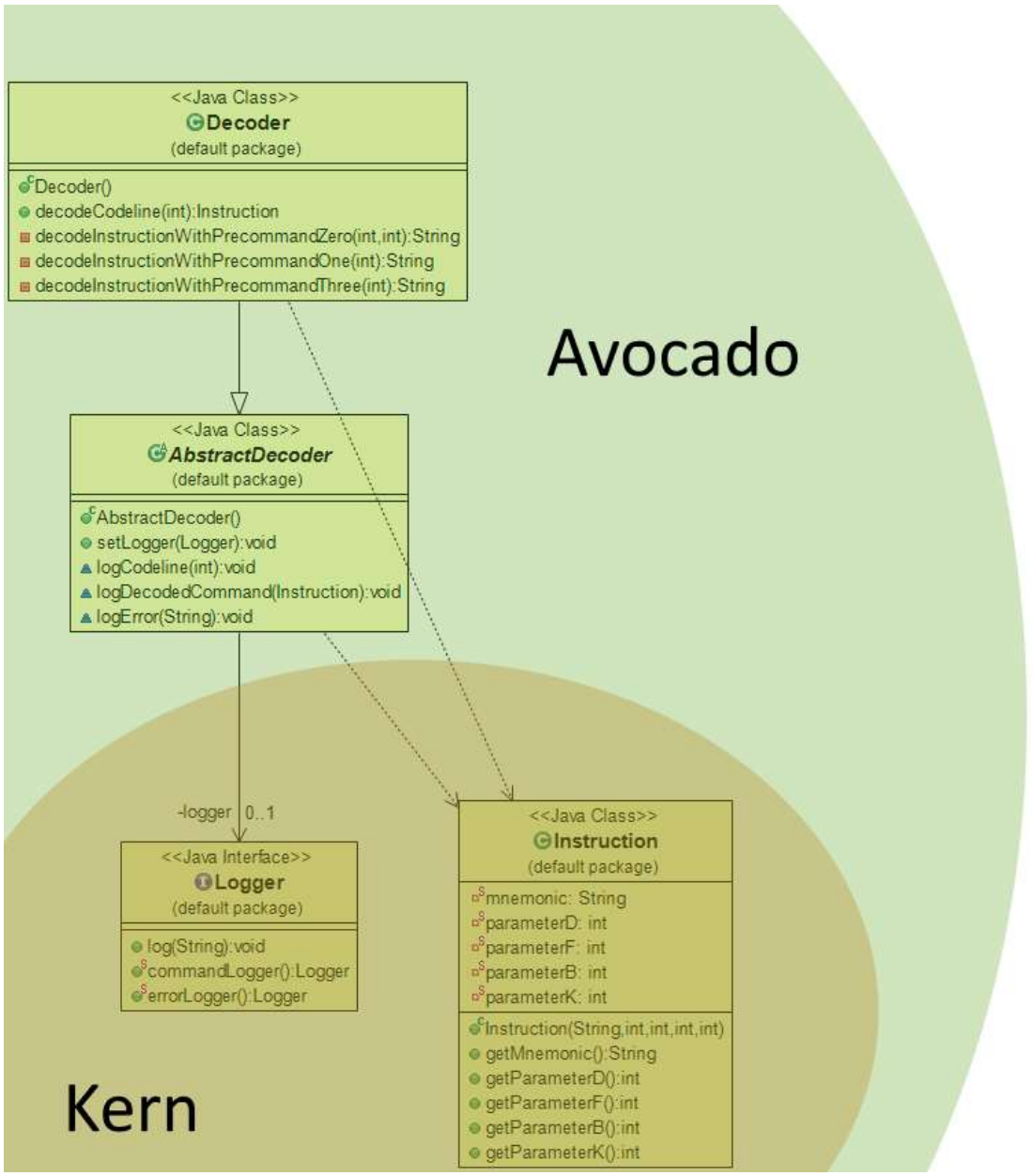
Die Clean Architecture auf das gesamte Projekt anzuwenden ist schwierig, da sie schwerwiegende Eingriffe in die Funktionalität bedeuten würde. Befände sich der Simulator im Praxiseinsatz, wäre dies auch definitiv eine notwendige Maßnahme aufgrund der vielen und schweren Verstöße gegen das Dependency Inversion Principle. Da das Projekt allerdings nicht im produktiven Betrieb ist und lediglich zu Anschauungszwecken entworfen und bearbeitet wurde, ist es einfacher und verständlicher, die Clean Architecture in einem Teilbereich des Projekts umzusetzen. Da die Processor-Klasse bereits einem groß angelegten Rework unterzogen wird, ist es sinnvoll, die dort entstehende Neustruktur nach dem Prinzip der Clean Architecture zu entwerfen.

Für dieses Prinzip gibt es verschiedene Namen, weit verbreitet ist dabei das Zwiebelmodell. Wie bei den Schalen einer Zwiebel wird dabei die Abhängigkeit der Klassen von innen nach außen gestaltet. Doch diese Analogie zeichnet nicht das volle Bild, weshalb das Avocado-Modell nach Simon König wahrscheinlich sprechender ist: im Inneren gibt es einen harten Kern, welcher außen von weicher formbarer Masse umgeben ist. Dieser harte Kern zeichnet sich dadurch aus, dass die dort angesiedelten Klassen im Laufe der Projektdauer nicht modifiziert werden – sie enthalten im Normalfall die grundlegenden Funktionalitäten wie zum Beispiel mathematische Operationen. Wie auch der Kern einer Avocado widerstehen sie jeder äußerlichen Einwirkung und behalten ihre Form und Struktur bei.

Das weiche Fruchtfleisch im äußeren Bereich der Avocado ist der Teil, der für den Genuss wirklich relevant ist. Dasselbe gilt für den weichen Teil im äußeren Bereich der Code-Architektur. Hier werden die Funktionalitäten aus dem Kern zur Anwendung gebracht; erst dadurch erhält das Projekt seine finale Ausgestaltung. In dieser äußeren Hülle finden sich gestalterische Elemente und Implementationen, welche leicht anpassbar sind und so stets die Möglichkeit bieten, das Projekt in eine neue Richtung zu lenken.

Der ursprüngliche Code des Simulators mutet allerdings eher Guacamole an als einer Avocado. Einen harten funktionalen Kern gibt es nicht; stattdessen hängen die Klassen lose miteinander zusammen anstelle, dass sie eine einheitliche Struktur formen. Um einen harten Kern zu schaffen, wurden essenzielle Funktionalitäten wie das Speichern und Transportieren von Mnemonics und den dazu gehörenden Operanden oder das Loggen der eingelesenen Codezeilen sowie den jeweiligen entschlüsselten Befehlen werden von Klassen im Kern übernommen. Die Klassen, die diese Funktionalität verwenden und darauf aufbauend die Umsetzung realisieren, werden in der äußeren formbaren Hülle aufgeführt.

Die folgende Seite zeigt ausschnittsweise, wie das Avocado-Modell für die Schaffung einer Clean Architecture herangezogen wurde. Aus Gründen der Übersichtlichkeit wurden die meisten Klassen des Simulators (wie zum Beispiel die Processor-Klasse) ausgenommen, diese lägen innerhalb weiterer Schichten außerhalb der des Decoders.


<<Java Class>>
Decoder
(default package)
Decoder()
decodeCodeline(int):Instruction
decodeInstructionWithPrecommandZero(int,int):String
decodeInstructionWithPrecommandOne(int):String
decodeInstructionWithPrecommandThree(int):String
Avocado
<<Java Class>>
AbstractDecoder
(default package)
AbstractDecoder()
setLogger(Logger):void
logCodeline(int):void
logDecodedCommand(Instruction):void
logError(String):void
-logger
0..1
<<Java Interface>>
Logger
(default package)
log(String):void
commandLogger():Logger
errorLogger():Logger
<<Java Class>>
Instruction
(default package)
mnemonic: String
parameterD: int
parameterF: int
parameterB: int
parameterK: int
Instruction(String,int,int,int,int)
getMnemonic():String
getParameterD():int
getParameterF():int
getParameterB():int
getParameterK():int
Kern

Unit Tests

Die erstellten Unit-Tests wurden für die bereits angesprochene neue Klasse Decoder erstellt und sind allesamt in der Tester-Klasse angesiedelt. Um ein Verständnis für die verwendeten Parameter zu bekommen, ist ebenfalls die Seite 56 der Dokumentation des PIC16F84 sehr nützlich. Mithilfe der dort aufgelisteten Bit-Codierungen wurden die Codezeilen für die Unittests erstellt.

Bei den Unit Tests für dieses Projekt wurden statt echten Codezeilen aus Beispielprogrammen fingierte Codezeilen genommen. Durch dieses Vortäuschen, auch Mocken genannt, wird eine kontrollierte Umgebung geschaffen und der zu testende Code wird isoliert. Anstatt dass eine Codezeile aus einem Programm eingelesen werden muss, wird also eine erstellt. Aufgrund dieses Erstellens wird die Anzahl möglicher Fehlerquellen minimiert – was genau das Ziel eines Unit-Tests ist. Für den folgenden Code sind also die Fehlerquellen lediglich darauf beschränkt, dass entweder das Decoding nicht funktioniert oder die Ergebnisausgabe mithilfe des Instruction-Objekts. Wichtig ist jedoch, dass die Tests die ATRIP-Regeln erfüllen:

```java
/*
 * Has already found a bug
 */
@Test
public void testDecoderWithNOP() {
    Decoder testDecoder = new Decoder();
    int line = 0b00000000000000;

    Instruction decodedInstruction = testDecoder.decodeCodeline(line);
    assertEquals(decodedInstruction.getCommand(), "NOP");
    assertEquals(decodedInstruction.getParameterB(), 0);
    assertEquals(decodedInstruction.getParameterF(), 0);
    assertEquals(decodedInstruction.getParameterD(), 0);
    assertEquals(decodedInstruction.getParameterK(), 0);
}

@Test
public void testDecoderWithBTFSC() {
    Decoder testDecoder = new Decoder();
    int precommand = 0b01;  //has to be 1
    int command = 0b10;     //has to be 2
    int b = 0b000;          //can be any 3-digit binary number
    int f = 0b0000000;      //can be any 7-digit binary number
    int line = (precommand << 12) + (command << 10) + (b << 7) + f;

    Instruction decodedInstruction = testDecoder.decodeCodeline(line);
    assertEquals(decodedInstruction.getCommand(), "BTFSC");
    assertEquals(decodedInstruction.getParameterB(), b);
    assertEquals(decodedInstruction.getParameterF(), f);
    assertEquals(decodedInstruction.getParameterD(), 0);
    assertEquals(decodedInstruction.getParameterK(), 0);
}
```

- **Automatic?**
  → Die Test laufen halbautomatisch. Nach dem manuellen Start laufen sie automatisch durch.
- **Thorough?** Die Erfüllung dieser Bedingung ist bei 35 Assembler-Befehlen zu umfangreich für einen Proof of Concept beziehungsweise Proof of Understanding.
  → Thorough wird nicht erfüllt, ist aber im Kontext der Aufgabenstellung vernachlässigbar.
- **Repeatable?**
  → Ja, der Test führt immer zu demselben Ergebnis.
- **Independent?** Der Idealfall ist nicht abgedeckt, da jeder Test auch am Instruction-Objekt scheitern kann. Zur Risikomitigation werden hierfür eigene Tests eingeführt.
  → Die Tests sind so weit wie realistisch möglich unabhängig voneinander.
- **Professional?** Diese Bedingung wird im obigen Screenshot nicht erfüllt.

Da die Professional-Anforderung nicht im sinnvollen Rahmen erfüllt wurde, mussten die Tests deutlich lesbarer programmiert werden:

```java
/*
 * Has already found a bug
 */
@Test
public void testDecoderWithNOP() {
    Decoder testDecoder = new Decoder();
    int line = 0b00000000000000;    //has to be zero
    int actualParameterD = 0;       //has to be zero
    int actualParameterF = 0;       //has to be zero
    int actualParameterB = 0;       //has to be zero
    int actualParameterK = 0;       //has to be zero

    Instruction decodedInstruction = testDecoder.decodeCodeline(line);
    assertEquals(decodedInstruction.getMnemonic(), "NOP");
    assertEquals(decodedInstruction.getParameterD(), actualParameterD);
    assertEquals(decodedInstruction.getParameterF(), actualParameterF);
    assertEquals(decodedInstruction.getParameterB(), actualParameterB);
    assertEquals(decodedInstruction.getParameterK(), actualParameterK);
}

@Test
public void testDecoderWithBTFSC() {
    Decoder testDecoder = new Decoder();
    int actualPrecommand = 0b01;        //has to be one
    int actualCommand = 0b10;           //has to be two
    int actualParameterB = 0b000;       //can be any 3-digit binary number
    int actualParameterF = 0b0000000;   //can be any 7-digit binary number
    int actualParameterD = 0;           //has to be zero
    int actualParameterK = 0;           //has to be zero
    int actualLine = (actualPrecommand << 12) + (actualCommand << 10) + (actualParameterB << 7) + actualParameterF;

    Instruction decodedInstruction = testDecoder.decodeCodeline(actualLine);
    assertEquals(decodedInstruction.getMnemonic(), "BTFSC");
    assertEquals(decodedInstruction.getParameterD(), actualParameterD);
    assertEquals(decodedInstruction.getParameterF(), actualParameterF);
    assertEquals(decodedInstruction.getParameterB(), actualParameterB);
    assertEquals(decodedInstruction.getParameterK(), actualParameterK);
}
```

Mit dieser Anpassung ist nun auch die Professional-Anforderung erfüllt und die ATRIP-Regeln werden ausreichend abgegolten. Dennoch ist weiterhin Luft nach oben, jeder Test nutzt aktuell alle Parameter, die insgesamt bei den Assembler-Befehlen möglich sind. Doch ein Befehl kann nicht alle Parameter gleichzeitig beinhalten, deshalb ist auch das Testen der nicht benötigten Operanden vollkommen unsinnig. Dadurch entstehen mögliche Problemstellen, die im eigentlichen Code nicht präsent sind und so unnötigerweise den Test scheitern lassen.

In der dritten Iteration der Unit-Tests werden nur noch die jeweilig benötigten Parameter in den Tests abgegolten. Außerdem wurden die Namen angepasst, um sprechender zu sein. Beim anschließenden Blick auf die Code/Branch Coverage fällt allerdings auf, dass für eine 100%-ige Code und Branch Coverage jeder einzelne Assembler-Befehl getestet werden muss. Für diesen Fall ist es sinnvoll, die Tests nach der abgeprüften Mnemonic zu benennen.

```java
/*
 * Has already found a bug
 * Tests decoder decoding on easy instruction without parameters
 */
@Test
public void testDecoderWithoutParameters() {
    Decoder testDecoder = new Decoder();

    String actualMnemonic = "NOP";
    int line = 0b00000000000000;    //has to be zero

    Instruction decodedInstruction = testDecoder.decodeCodeline(line);
    assertEquals(decodedInstruction.getMnemonic(), actualMnemonic);
}

/*
 * Tests decoder decoding on a more complex instruction with parameters B and F
 */
@Test
public void testDecoderWithParametersBF() {
    Decoder testDecoder = new Decoder();

    String actualMnemonic = "BTFSC";
    int actualPrecommand = 0b01;        //has to be one
    int actualCommand = 0b10;           //has to be two (10 in binary)
    int actualParameterF = 0b0000001;   //can be any 7-digit binary number
    int actualParameterB = 0b001;       //can be any 3-digit binary number

    int actualLine = (actualPrecommand << 12) + (actualCommand << 10) + (actualParameterB << 7) + actualParameterF;

    Instruction decodedInstruction = testDecoder.decodeCodeline(actualLine);
    assertEquals(decodedInstruction.getMnemonic(), actualMnemonic);
    assertEquals(decodedInstruction.getParameterF(), actualParameterF);
    assertEquals(decodedInstruction.getParameterB(), actualParameterB);
}
```

Bezüglich der Code Coverage für die Klasse Decoder ist diese in weiten Teilen noch nicht erfüllt. Da für jeden möglichen Befehl des Microcontrollers eine eigene Bedingung existiert, werden bei den stichprobenartigen Abfragen der Unit-Tests die meisten Branches nicht abgegolten. Daraus resultieren viele rote Codezeilen.

Zur Beseitigung dieses Missstands wird es notwendig sein, für jeden einzelnen Assembler-Befehl einen eigenen Unit-Test zu schreiben. Diese sehen nahezu identisch aus zu den Tests im Screenshot und denen im Code, sodass sie durch Copy-Paste und Anpassung der Parameter recht einfach erstellt werden können. Da diese Dokumentation lediglich ein Proof of Concept beziehungsweise Proof of Understanding sein soll, werden weitere Unit-Tests in diesem Bereich nicht notwendig sein.

```
private String decodeInstructionWithPrecommandZero(int command, int payload) {
    int d = payload >> 7 & 0x0001;
    String output = "NOP";

    switch (command) {
    case 7: output = "ADDWF"; break;
    case 5: output = "ANDWF"; break;
    case 1:
        if (d == 1) {output = "CLRF";}
        else        {output = "CLRW";}
        break;
    case 9:  output = "COMF";   break;
    case 3:  output = "DECF";   break;
    case 11: output = "DECFSZ"; break;
    case 10: output = "INCF";   break;
    case 15: output = "INCFSZ"; break;
    case 4:  output = "IORWF";  break;
    case 8:  output = "MOVF";   break;
    case 0:
        if (d == 1) { output = "MOVWF";}
        else {
            switch (payload) {
            case 0b01100100: output = "CLRWDT"; break;
            case 0b00001001: output = "RETFIE"; break;
            case 0b00001000: output = "RETURN"; break;
            case 0b01100011: output = "SLEEP";  break;
            default:         output = "NOP";    break;
            }
        }
        break;
    case 13: output = "RLF"; break;
    case 12: output = "RRF"; break;
    case 2:  output = "SUBWF"; break;
    case 14: output = "SWAPF"; break;
    case 6:  output = "XORWF"; break;
    }
    return output;
}
```

Im Gegensatz zur Befehlszuordnung wird der erste Teil des Decoders nahezu vollständig abgedeckt. Da bei den Unit-Tests jeder Precommand mindestens einmal mitcodiert wird, Lediglich an einer Stelle findet sich eine rote Zeile – ebenfalls bei der Zuordnung eines konkreten Befehls. Auch diese rote Zeile lässt sich mit dem Duplizieren und Anpassen eines bestehenden Unit-Tests leicht beseitigen.

Für die finale Version des Projekts wurde dies umgesetzt, die Funktion decodeCodeline hat also eine 100%-ige Code und Branch Coverage. Es wird noch ein fehlender Branch angezeigt, damit wird auf den nicht implementiert ELSE-Zweig verwiesen. Rein technisch können aber keine Probleme auftreten, da sich der Zahlenbereich aufgrund der binären Verundung nicht außerhalb der abgefragten Werte bewegt. Zur Sicherheit wurde für die finale Version noch eine Illegal-Argument-Exception implementiert.

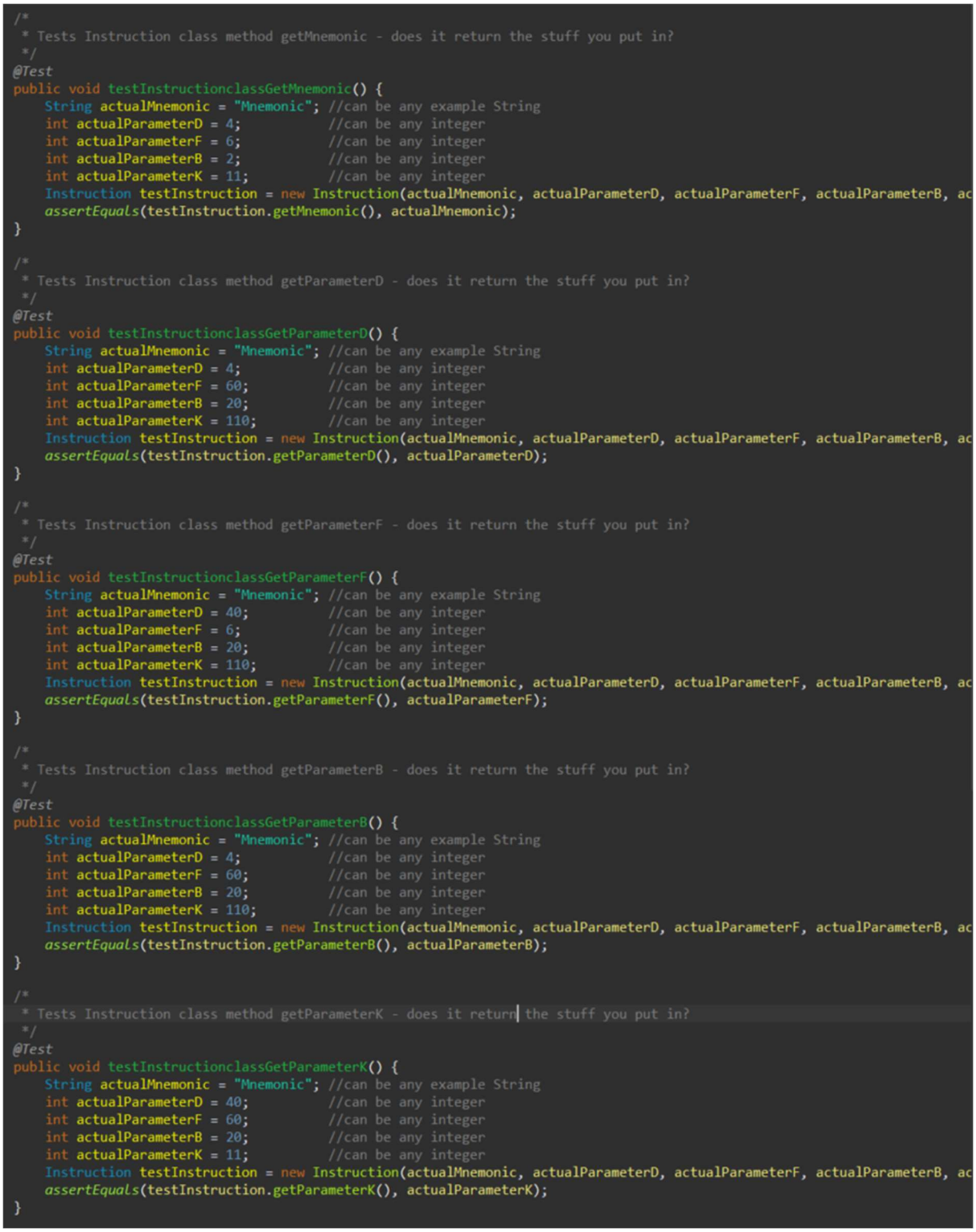
```
// Decode instruction into Assembler command and its parameters
protected Instruction decodeCodeline(int line) {
    System.out.println("Command " + Integer.toHexString(line));
    String assemblerCommand = "";
    int d = 0;
    int f = 0;
    int b = 0;
    int k = 0;

    int precommand = (line >> 12) & 0x0003;

    if (precommand == 0) {
        int command = (line >> 8) & 0x000f;
        int payload = line & 0x00ff;

        d = payload >> 7 & 0x0001;
        f = payload & 0x07F;
        assemblerCommand = decodeInstructionWithPrecommandZero(command, payload);
    }
    else if (precommand == 1) {
        int command = (line >> 10) & 0x0003;
        int payload = line & 0x03ff;

        b = (payload >> 7) & 0x0007;
        f = payload & 0x007f;
        assemblerCommand = decodeInstructionWithPrecommandOne(command);
    }
    else if (precommand == 2) {
        int command = (line >> 11) & 0x0001;

        k = line & 0x07ff;
        if      (command == 0)  {
            assemblerCommand = "CALL"; }
        else if (command == 1)  {
            assemblerCommand = "GOTO"; }
    }
    else if (precommand == 3) {
        int command = (line >> 8) & 0x000f;

        k = line & 0x00ff;
        assemblerCommand = decodeInstructionWithPrecommandThree(command);
    }

    Instruction fullInstruction = new Instruction(assemblerCommand, d, f, b, k);
    return fullInstruction;
}
```

Auch die Klasse Instruction wurde in den Unit-Tests abgedeckt. Da sie eine essenzielle Bedingung für die Erfüllung der anderen Unit-Tests darstellt, ist es sinnvoll, sie getrennt zu testen. Die oben bereits angesprochene Verletzung der ATRIP-Regel wird dadurch mitigiert. Scheitern die Decoder-Tests aber nicht die Instruction-Tests, liegt die Fehlerquelle im Decoder. Scheitert der Instruction-Test jedoch ebenfalls, kann die Fehlerquelle an beiden Stellen liegen, sollte aufgrund der geringeren Klassenkomplexität zuerst Instruction untersucht/repariert werden.

```java
public class Instruction {

    private static String mnemonic; //mnemonic is the "readable" version of the Assembler command
    private static int parameterD;
    private static int parameterF;
    private static int parameterB;
    private static int parameterK;

    public Instruction(String givenMnemonic, int givenParameterD, int givenParameterF, int givenParameterB, int givenParameterK) {
        Instruction.mnemonic = givenMnemonic;
        Instruction.parameterD = givenParameterD;
        Instruction.parameterF = givenParameterF;
        Instruction.parameterB = givenParameterB;
        Instruction.parameterK = givenParameterK;
    }

    public String getMnemonic() {
        return mnemonic;
    }

    public int getParameterD() {
        return parameterD;
    }

    public int getParameterF() {
        return parameterF;
    }

    public int getParameterB() {
        return parameterB;
    }

    public int getParameterK() {
        return parameterK;
    }

}
```

```java
/*
 * Tests Instruction class - does it output the stuff you put in?
 */
@Test
public void testInstructionclassReturns() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 4;           //can be any integer
    int actualParameterF = 6;           //can be any integer
    int actualParameterB = 2;           //can be any integer
    int actualParameterK = 11;          //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, actualParameterK);
    assertEquals(testInstruction.getMnemonic(), actualMnemonic);
    assertEquals(testInstruction.getParameterD(), actualParameterD);
    assertEquals(testInstruction.getParameterF(), actualParameterF);
    assertEquals(testInstruction.getParameterB(), actualParameterB);
    assertEquals(testInstruction.getParameterK(), actualParameterK);
}
```

Der aktuell noch zusammengefasste Unit-Test der Klasse Instruction bieten die perfekte Grundlage, um den Idealfall der Independent-Bedingung abzugelten: wenn für jede einzelne get-Methode ein eigener Unit-Test existiert, wird außer dem Konstruktor jede einzelne Komponente der Klasse in einem exklusiven Unit-Test abgeprüft. Die finale Version der Tests befindet sich auf der nächsten Seite.



```java
/*
 * Tests Instruction class method getMnemonic - does it return the stuff you put in?
 */
@Test
public void testInstructionclassGetMnemonic() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 4;           //can be any integer
    int actualParameterF = 6;           //can be any integer
    int actualParameterB = 2;           //can be any integer
    int actualParameterK = 11;          //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, ac
    assertEquals(testInstruction.getMnemonic(), actualMnemonic);
}

/*
 * Tests Instruction class method getParameterD - does it return the stuff you put in?
 */
@Test
public void testInstructionclassGetParameterD() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 4;           //can be any integer
    int actualParameterF = 60;          //can be any integer
    int actualParameterB = 20;          //can be any integer
    int actualParameterK = 110;         //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, ac
    assertEquals(testInstruction.getParameterD(), actualParameterD);
}

/*
 * Tests Instruction class method getParameterF - does it return the stuff you put in?
 */
@Test
public void testInstructionclassGetParameterF() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 40;          //can be any integer
    int actualParameterF = 6;           //can be any integer
    int actualParameterB = 20;          //can be any integer
    int actualParameterK = 110;         //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, ac
    assertEquals(testInstruction.getParameterF(), actualParameterF);
}

/*
 * Tests Instruction class method getParameterB - does it return the stuff you put in?
 */
@Test
public void testInstructionclassGetParameterB() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 4;           //can be any integer
    int actualParameterF = 60;          //can be any integer
    int actualParameterB = 20;          //can be any integer
    int actualParameterK = 110;         //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, ac
    assertEquals(testInstruction.getParameterB(), actualParameterB);
}

/*
 * Tests Instruction class method getParameterK - does it return the stuff you put in?
 */
@Test
public void testInstructionclassGetParameterK() {
    String actualMnemonic = "Mnemonic"; //can be any example String
    int actualParameterD = 40;          //can be any integer
    int actualParameterF = 60;          //can be any integer
    int actualParameterB = 20;          //can be any integer
    int actualParameterK = 11;          //can be any integer
    Instruction testInstruction = new Instruction(actualMnemonic, actualParameterD, actualParameterF, actualParameterB, ac
    assertEquals(testInstruction.getParameterK(), actualParameterK);
}
```

Die abgeschnittenen Parameter in den Zeilen mit der Erstellung des Instruction-Objekts sind identisch zu den Zeilen aus dem vorherigen Screenshot. Die unvollständige Darstellung ist dem Seitenverhältnis des Monitors geschuldet, im Hochformat sind die Zeilen zu lang, um vollständig dargestellt zu werden.

Die einhundertprozentige Code Coverage in der Klasse Instruction wird beibehalten.

Refactoring

Refactorings sind graduelle Verbesserungen des Codes, durch die auch alter Code immer weiter verbessert und ausgebaut werden kann. In einem Projekt, das regelmäßig refactort wird, ist es nahezu unmöglich, dass „Code-Ungetüme“ entstehen. Auch in der Software-Entwicklung gibt es provisorische Lösungen, die am Ende langfristig bestehen bleiben. Dazu werden durch stetig verändernde Anforderungen und neu hinzugefügte Features Implementationen, die auf kleinem Maßstab elegant waren, mehr und mehr zu komplexen Konstrukten, bei denen eine andere Herangehensweise eleganter ist.

Beim Refactoring werden genau solche problematischen Codestellen (auch Codesmell genannt) ausfindig gemacht und durch elegante, zukunftsfähige Lösungen ersetzt. Selbst Code, der bereits einem Refactoring unterzogen wurde, kann eine erneute Überarbeitung benötigen.

Ein Beispiel für die erwähnten Lösungen im kleinen Maßstab stellen IF-Statements dar. Während sie bei nur wenigen abgeprüften Bedingungen elegant sind, aber bei zu vielen Abfragen zu komplizierten Ketten mutieren. Im Decoding-Bereich des Controllers war eine solche IF-Verkettung angesiedelt. In dieser Kette wurde eine Variable auf verschiedene Werte überprüft – ein Ablauf, der deutlich einfacher in einer SWITCH CASE Struktur abzubilden ist. Außerdem enthält die IF-Kette für jeden einzelnen decodierten Befehl einen Kommentar, wie dieser Befehl heißt – obwohl diese Information bereits aus dem Aufruf des entsprechenden Befehls ersichtlich wird. Sprechender Code benötigt keine Kommentare, also kann auch diese Information entfernt werden.

ALT:

```
if (command == 7) {
        // ADDWF
        this.ADDWF(f, d);
} else if (command == 5) {
        // ANDWF
        this.ANDWF(f, d);
} else if (command == 1) {
        if (d == 1) {
                // CLRF
                this.CLRF(f);
        } else {
                // CLRW
                this.CLRW(w);
        }
} else if (command == 9) {
        // COMF
        this.COMF(f, d);
} else if (command == 3) {
        // DECF
        this.DECF(f, d);
} else if (command == 11) {
        // DECFSZ
        this.DECFSZ(f, d);
```

NEU:

```
switch (command) {
case 7:
    this.ADDWF(f, d);
    break;
case 5:
    this.ANDWF(f, d);
    break;
case 1:
    if (d == 1) {this.CLRF(f);}
    else        {this.CLRW();}
    break;
case 9:
    this.COMF(f, d);
    break;
case 3:
    this.DECF(f, d);
    break;
case 11:
    this.DECFSZ(f, d);
    break;
```

Was ebenfalls auffällt, ist das Fehlen eines Default-Statements und die falsche numerische Reihenfolge der Überprüfungen. Diese beiden Entscheidungen sind allerdings bewusst getroffen worden. Die falsche numerische Reihenfolge folgt aus der Reihenfolge der Befehlsüberprüfung. Die Befehle werden in der Reihenfolge abgeprüft, in der sie in der Dokumentation des PIC-Microcontrollers gelistet werden. Dadurch wird zwar die Verständlichkeit des reinen Codes reduziert, wie in der Einleitung aber bereits erwähnt bleibt sie dafür erhalten, wenn der Code mit der Dokumentation des Microcontrollers aufgearbeitet wird.

Das Fehlen des Default-Statement resultiert daraus, dass der gesamte mögliche Wertebereich bereits abgedeckt ist. Beim Umzug der Decoding-Funktionalität in eine eigene Klasse wurde allerdings auch diese Unreinheit beseitigt und der default case nach Abprüfung aller Parameter ist das Eintragen eines Fehlers in dem Konsolen-Log und das Werfen einer Illegal Argument Exception.

Eine Besonderheit stellt jedoch die Zuordnung bei einem Befehl mit dem Precommand 3 (also 11 in binärer Schreibweise) dar: in diesem Fall variiert die Position und Länge der Command-Bits, sodass mehrere Bitshifts zur korrekten Erkennung notwendig sind. Eine Überprüfung der command-Variable mithilfe eines SWITCH CASE wäre unübersichtlich, was die Lesbarkeit des Codes deutlich mehr beeinträchtigen würde als die aktuelle Lösung mit verketteten IF-Statements. Die Verkettung wird also zum Erhalt einer besseren Code-Lesbarkeit (in diesem speziellen Kontext) beibehalten.

```
else if (precommand == 3) {
    int command = (line >> 8) & 0x000f;
    int k = line & 0x00ff;

    if (command == 8) {
        this.IORLW(k);
    }
    else if ((command >> 1) == 7) {
        this.ADDLW(k);
    }
    else if (command == 9) {
        this.ANDLW(k);
    }
    else if ((command >> 2) == 0) {
        this.MOVLW(k);
    }
    else if ((command >> 2) == 1) {
        this.RETLW(k);
    }
    else if (command == 10) {
        this.XORLW(k);
    }
    else if ((command >> 1) == 6) {
        this.SUBLW(k);
    }
    else {
        this.NOP();
    }
}
```

Die Decodier-Anweisung stellt aber noch einen weiteren Codesmell dar, denn sie ist inklusive Kommentare über 150 Zeilen lang. Damit ist sie deutlich länger als für eine verständliche Aufarbeitung förderlich ist. Die Lösung ist die Auslagerung der Command-Decodierung auf zusätzliche Methoden, die den Command in Abhängigkeit vom Precommand zuordnen. Für jeden Precommand wird eine neue Methode geschaffen, in der dem Bitcode die korrekte Mnemonic zugeordnet wird.

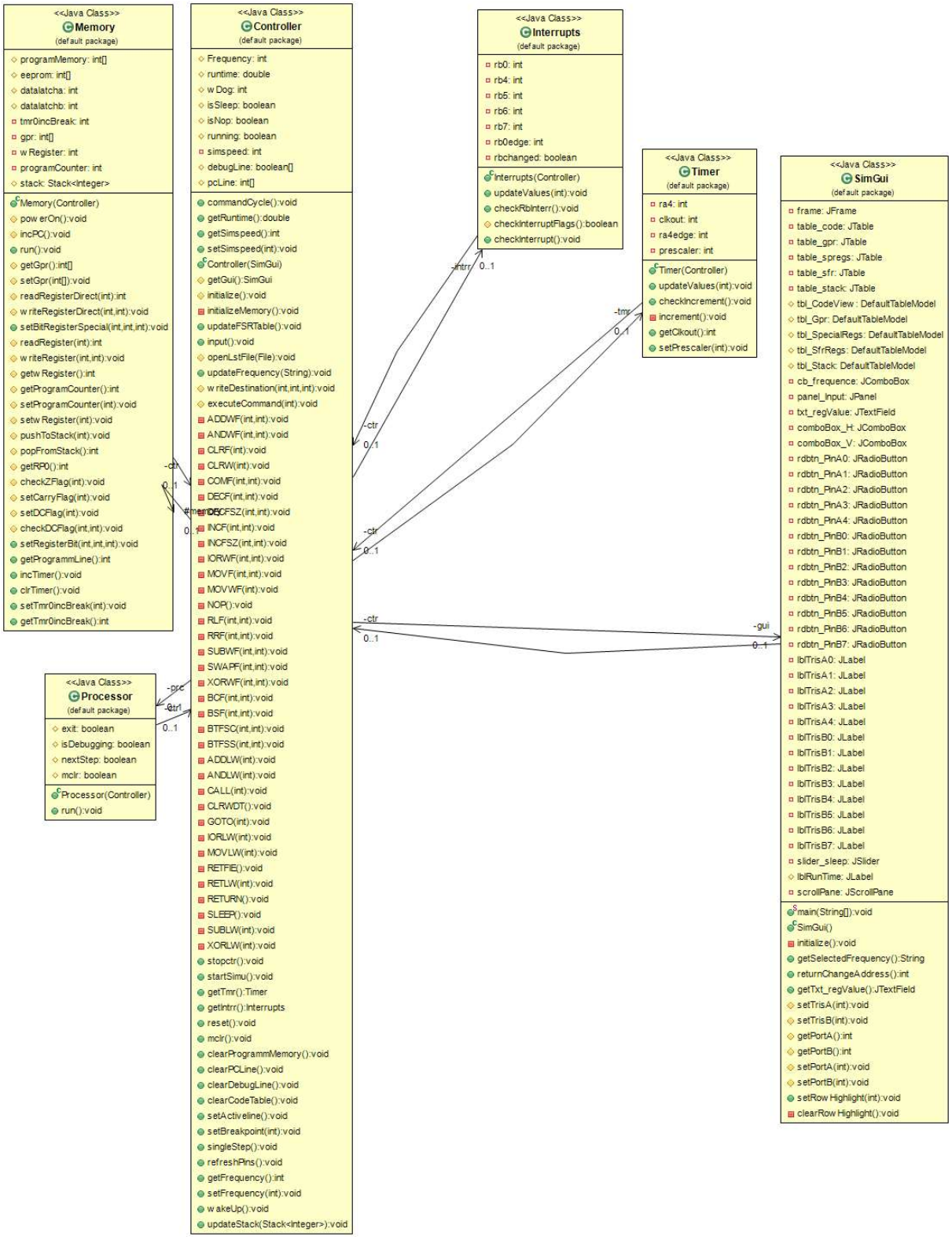
Viele dieser Refactorings sind im gleichen Zug geschehen wie die Entkoppelung der Opcode-Decodierung von der Controller-Klasse. Dieses Auslagern stellt ebenfalls ein Refactoring für sich dar, da die Klasse Controller sehr umfangreich war – und immer noch ist.

Um die Kommunikation der Decoder-Klasse mit dem Rest des Projekts zu ermöglichen wurde eine neue Hilfsklasse eingeführt, die Instruction. Sie hat mehrere statische Variablen, unter anderem einen String, in dem die entschlüsselte Mnemonic abgespeichtert wird und dazu noch vier Integer, je einer für jeden der auf Seite 56 der Dokumentation festgelegten Operanden D, F, B und K. Die Werte für diese Variablen können nur über den Konstruktor festgelegt werden, zum Auslesen existiert jeweils eine get-Methode.

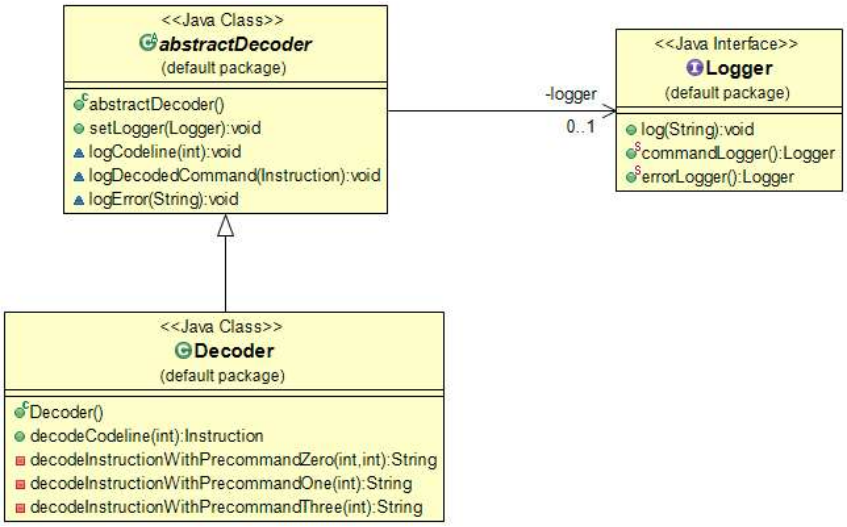
Entwurfsmuster

Entwurfsmuster (engl. Design Patterns) helfen bei der Umsetzung spezieller Programmabläufe und sind darauf ausgelegt, einfache Erweiterungsmöglichkeiten zu bieten. Sie stellen oftmals ein spezielles Abhängigkeitsverhältnis von Klassen und abstrakten Klassen zueinander dar. Durch diese spezifischen Beziehungen entstehen Konstrukte, deren Zusammenwirken beinahe magisch wirkt.

Zu Beginn der Transformation sah die Klassenstruktur des Projekts wie folgt aus:



Eingesetzt wurde das Bridge Pattern, welches zwar nicht in den Vorlesungen besprochen wurde, aber für den Konsolen-Log der eingelesenen Codezeile und der dekodierten Befehle perfekt geeignet ist. Das UML-Diagramm des Patterns sieht wie folgt aus:



Wichtig ist dabei anzumerken, dass die Implementierungen des Logger-Interfaces bereits im Interface selbst festgelegt werden. Ermöglicht wird dies durch das Java-Feature, funktionale Interfaces zu erstellen.

Durch die Brücke zwischen abstractDecoder und Logger werden die konkreten Implementierungen Decoder und commandLogger (beziehungsweise ErrorLogger) voneinander getrennt und können so unabhängig verwaltet werden. So kann eine alternative Implementierung des Decoders problemlos alle Logger nutzen und ein neuer Logger wird sofort allen Decoder-Implementierungen zur Verfügung gestellt. Die Umsetzung des Bridge Patterns ähnelt dabei stark dem Adapter Pattern. Die konkrete Umsetzung in diesem Projekt stellt einen Sonderfall dar, weil sie dem Object Adapter Pattern folgt.

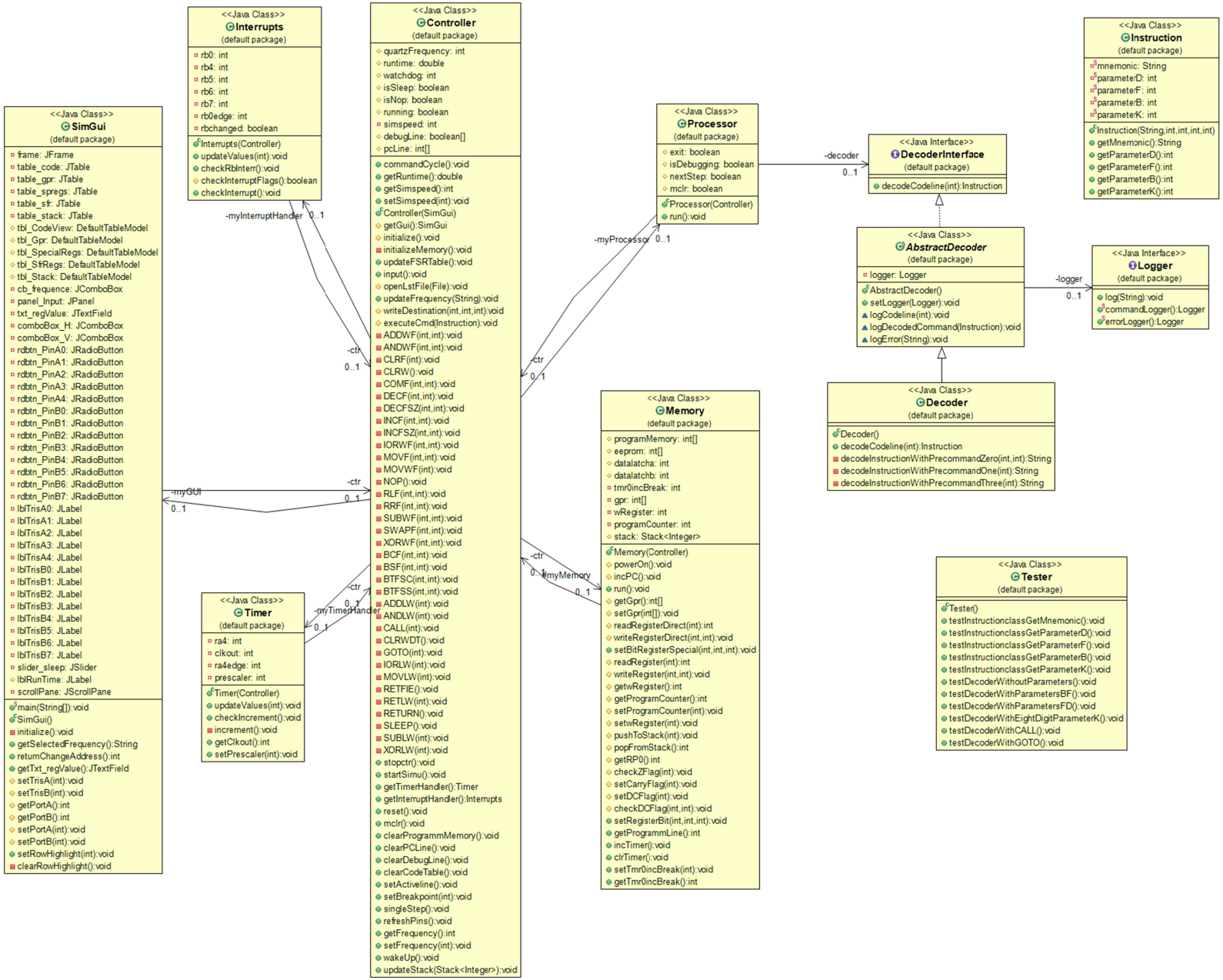
Der Logger ist ebenfalls für die neu eingeführten Fehler-Fälle innerhalb des Decoders zuständig. Über den errorLogger wird ausgegeben, wenn ein Argument trotz aller mathematischen Widersprüche außerhalb des festgelegten Wertebereichs liegt.

Fazit

Obwohl der Simulator weiterhin viele Problemstellen besitzt, konnten einige davon beseitigt werden durch gezielte Aufarbeitung, Anwendung von Programmierprinzipien, Planung einer Clean Architecture und Erstellung effektiver Unit Tests. Eine Übertragung der angewendeten Maßnahmen auf das gesamte Projekt kann also den uneleganten Ausgangscode umwandeln in eine Programmstruktur, die moderne Standards sowie die damit verknüpften hohen Ansprüche erfüllt.

Am Kern rund um die Controller-Klasse hat sich beim Blick auf das UML-Diagramm scheinbar nur wenig getan. Dennoch sind aufgrund der Implementierung neuer Funktionalitäten diese Klassen entlastet worden – und die unternommenen Neustrukturierungen zeigen eindeutig, dass auch die noch bestehenden Problemstellen in guten Code umgewandelt werden können.

Die abschließende Projektstruktur befindet sich als UML-Diagramm auf der letzten Seite dieser Dokumentation.


<<Java Class>>
SimGui
(default package)
frame: JFrame
table_code: JTable
table_gpr: JTable
table_spregs: JTable
table_sfr: JTable
table_stack: JTable
tbl_CodeView: DefaultTableModel
tbl_Gpr: DefaultTableModel
tbl_SpecialRegs: DefaultTableModel
tbl_SfrRegs: DefaultTableModel
tbl_Stack: DefaultTableModel
cb_frequence: JComboBox
panel_Input: JPanel
txt_regValue: JTextField
comboBox_H: JComboBox
comboBox_V: JComboBox
rdbtn_PinA0: JRadioButton
rdbtn_PinA1: JRadioButton
rdbtn_PinA2: JRadioButton
rdbtn_PinA3: JRadioButton
rdbtn_PinA4: JRadioButton
rdbtn_PinB0: JRadioButton
rdbtn_PinB1: JRadioButton
rdbtn_PinB2: JRadioButton
rdbtn_PinB3: JRadioButton
rdbtn_PinB4: JRadioButton
rdbtn_PinB5: JRadioButton
rdbtn_PinB6: JRadioButton
rdbtn_PinB7: JRadioButton
lblTrisA0: JLabel
lblTrisA1: JLabel
lblTrisA2: JLabel
lblTrisA3: JLabel
lblTrisA4: JLabel
lblTrisB0: JLabel
lblTrisB1: JLabel
lblTrisB2: JLabel
lblTrisB3: JLabel
lblTrisB4: JLabel
lblTrisB5: JLabel
lblTrisB6: JLabel
lblTrisB7: JLabel
slider_sleep: JSlider
lblRunTime: JLabel
scrollPane: JScrollPane
main(String[]):void
SimGui()
initialize():void
getSelectedFrequency():String
returnChangeAddress():int
getTxt_regValue():JTextField
setTrisA(int):void
setTrisB(int):void
getPortA():int
getPortB():int
setPortA(int):void
setPortB(int):void
setRowHighlight(int):void
clearRowHighlight():void
<<Java Class>>
Interrupts
(default package)
rb0: int
rb4: int
rb5: int
rb6: int
rb7: int
rb0edge: int
rbchanged: boolean
Interrupts(Controller)
updateValues(int):void
checkRbInterr():void
checkInterruptFlags():boolean
checkInterrupt():void
<<Java Class>>
Controller
(default package)
quartzFrequency: int
runtime: double
watchdog: int
isSleep: boolean
isNop: boolean
running: boolean
simspeed: int
debugLine: boolean[]
pcLine: int[]
commandCycle():void
getRuntime():double
getSimspeed():int
setSimspeed(int):void
Controller(SimGui)
getGui():SimGui
initialize():void
initializeMemory():void
updateFSRTable():void
input():void
openLstFile(File):void
updateFrequency(String):void
writeDestination(int,int,int):void
executeCmd(Instruction):void
ADDWF(int,int):void
ANDWF(int,int):void
CLRF(int):void
CLRW():void
COMF(int,int):void
DECF(int,int):void
DECFSZ(int,int):void
INCF(int,int):void
INCFSZ(int,int):void
IORWF(int,int):void
MOVF(int,int):void
MOVWF(int):void
NOP():void
RLF(int,int):void
RRF(int,int):void
SUBWF(int,int):void
SWAPF(int,int):void
XORWF(int,int):void
BCF(int,int):void
BSF(int,int):void
BTFSC(int,int):void
BTFSS(int,int):void
ADDLW(int):void
ANDLW(int):void
CALL(int):void
CLRWDT():void
GOTO(int):void
IORLW(int):void
MOVLW(int):void
RETFIE():void
RETLW(int):void
RETURN():void
SLEEP():void
SUBLW(int):void
XORLW(int):void
stopctr():void
startSimu():void
getTimerHandler():Timer
getInterruptHandler():Interrupts
reset():void
mclr():void
clearProgrammMemory():void
clearPCLine():void
clearDebugLine():void
clearCodeTable():void
setActiveline():void
setBreakpoint(int):void
singleStep():void
refreshPins():void
getFrequency():int
setFrequency(int):void
wakeUp():void
updateStack(Stack<Integer>):void
<<Java Class>>
Processor
(default package)
exit: boolean
isDebugging: boolean
nextStep: boolean
mclr: boolean
Processor(Controller)
run():void
<<Java Interface>>
DecoderInterface
(default package)
decodeCodeline(int):Instruction
<<Java Class>>
Instruction
(default package)
mnemonic: String
parameterD: int
parameterF: int
parameterB: int
parameterK: int
Instruction(String,int,int,int,int)
getMnemonic():String
getParameterD():int
getParameterF():int
getParameterB():int
getParameterK():int
<<Java Class>>
AbstractDecoder
(default package)
logger: Logger
AbstractDecoder()
setLogger(Logger):void
logCodeline(int):void
logDecodedCommand(Instruction):void
logError(String):void
<<Java Interface>>
Logger
(default package)
log(String):void
commandLogger():Logger
errorLogger():Logger
<<Java Class>>
Decoder
(default package)
Decoder()
decodeCodeline(int):Instruction
decodeInstructionWithPrecommandZero(int,int):String
decodeInstructionWithPrecommandOne(int):String
decodeInstructionWithPrecommandThree(int):String
<<Java Class>>
Memory
(default package)
programMemory: int[]
eeprom: int[]
datalatcha: int
datalatchb: int
tmr0incBreak: int
gpr: int[]
wRegister: int
programCounter: int
stack: Stack<Integer>
Memory(Controller)
powerOn():void
incPC():void
run():void
getGpr():int[]
setGpr(int[]):void
readRegisterDirect(int):int
writeRegisterDirect(int,int):void
setBitRegisterSpecial(int,int,int):void
readRegister(int):int
writeRegister(int,int):void
getwRegister():int
getProgramCounter():int
setProgramCounter(int):void
setwRegister(int):void
pushToStack(int):void
popFromStack():int
getRP0():int
checkZFlag(int):void
setCarryFlag(int):void
setDCFlag(int):void
checkDCFlag(int,int):void
setRegisterBit(int,int,int):void
getProgrammLine():int
incTimer():void
clrTimer():void
setTmr0incBreak(int):void
getTmr0incBreak():int
<<Java Class>>
Timer
(default package)
ra4: int
clkout: int
ra4edge: int
prescaler: int
Timer(Controller)
updateValues(int):void
checkIncrement():void
increment():void
getClkout():int
setPrescaler(int):void
<<Java Class>>
Tester
(default package)
Tester()
testInstructionclassGetMnemonic():void
testInstructionclassGetParameterD():void
testInstructionclassGetParameterF():void
testInstructionclassGetParameterB():void
testInstructionclassGetParameterK():void
testDecoderWithoutParameters():void
testDecoderWithParametersBF():void
testDecoderWithParametersFD():void
testDecoderWithEightDigitParameterK():void
testDecoderWithCALL():void
testDecoderWithGOTO():void
-myInterruptHandler
0..1
-ctr
0..1
-myProcessor
-decoder
-logger
-myGUI
-myTimerHandler
-myMemory