Review Excercise - Bookkeeper

June 12, 2012

Certified Tester Praktikum 3

Jens Schaa, Lucas Jenss, Denis Fleischhauer

1 Code anomalies and dead code

File: `src/de/harper_hall/keeper/spells/elemental/base/area_elemental_curses/AreaElementalCursesSpellList.java`

The mentioned (and several other) files contain definitions for spells, weapons etc. used within the game. This is not an optimal solution, as it requires a lot of duplicate code. Every spell definition, for example, has the following structure:

```
addSpell(
  new GenericSpellDescription(
    ... lots of parameters ...
  )
);
```

Such a structure can be recreated very easily in a data format such as `JSON`:

```
[ // Array of spells
    [ ... lots of parameters ...], // A single spell
    ...
    [ ... lots of parameters ...]
]
```

Outsourcing this data has the benefit of not being implementation dependant, that is, the data could easily be read and proccessed for different purposes without virtually any effort.

File: `src/de/harper_hall/keeper/tables/StatBonuses.java`

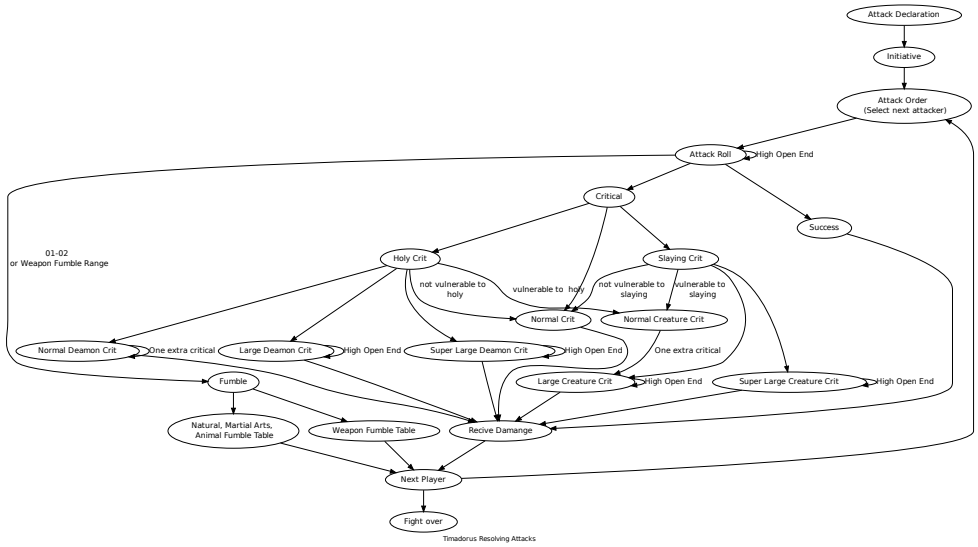
The methods `getStatBonus` and `getDevPts` have exactly the same implementation!

File: `src/de/harper_hall/keeper/classes/SkillDefinition.java`

The method with the signature `public Skill getSkill() throws SkillInvokationException` is defined twice within the class `SkillDefinition`, and it is not clear why. Also, their implementations only differ in a single line.

2 Testcases with coverage criteria

2.1 Timadorus Attack Graph



2.2 Test cases

Inputs specified in this order:

1. Weapon
2. Target
3. Rolls

In:

1. Sword,
2. Dog,
3. Fumble

Out: Attacker stumble

In:

1. Martial Art Fist Punch,
2. Dog,
3. Fumble

Out: Attacker stumble

In:

1. Sword,
2. Dog,
3. 2x Success

Out: Dog revices wounds

In:

1. Sword of Dog Slaying,
2. Dog,
3. Critical

Out: Dog revices wounds

In:

1. Sword of Dog Slaying,
2. Fisch,
3. Critical

Out: Fisch revices wounds

In:

1. Sword of Horse Slaying,
2. Horse,
3. 2x Critical

Out: Horse revices wounds

In:

1. Sword of Wale Slaying,
2. Wale,
3. 2x Critical

Out: Wale revices wounds

In:

1. Holy Sword,
2. Dog,
3. Critical

Out: Dog revices wounds

In:

1. Holy Sword,
2. Minor Deamon,
3. 2x Critical

Out: Minor Deamon revices wounds

In:

1. Holy Sword,
2. Big Deamon,
3. 2x Critical

Out: Big Deamon revices wounds

In:

1. Holy Sword,
2. Boss Deamon,
3. 2x Critical

Out: Boss Deamon revices wounds

3 LCOM

```
Klasse: CharCreatorTest
Instanzvariablen: creator
Methodenpaare:
    loadStats()
    checkTots() --> mit

    loadStats()
    checkPots() --> mit

    checkPots()
    testCharacterCreation() --> mit

LCOM = 0-ohne - 3-mit = 0

Klasse: KieperCharacterLoaderTest
Instanzvariablen: keine
Methodenpaare:
    testLoadSampleCharacter()
    testLoadNoClassSampleCharacter() --> ohne

LCOM = 1-ohne - 0-mit = 1

Klasse: RaceBaseTest
Instanzvariablen: keine
Methodenpaare:
    testRaceBaseInit()
    testRaceValueDump() --> ohne

LCOM = 1-ohne - 0-mit = 1

Klasse: AttackProcessTest
Instanzvariablen: aragorn, baromir
Methodenpaare:
    setUp()
    testInitialResolution() --> ohne

    setUp()
    testAttackResolution() --> mit

    testInitialResolution()
    testAttackResolution() --> ohne

LCOM = 2-ohne - 1-mit = 1

Klasse: KeeperCharacterBaseTest
Instanzvariablen:   lvl0Pies            lvl1Pies            lvl2Pies
                    lvl0CostExpert      lvl1CostExpert      lvl2CostExpert
                    lvl0RatingExpert    lvl1RatingExpert    lvl2RatingExpert
                    rolls               character
```

```
Methodenpaare:
    loadStats()
    prepChar() --> ohne

    loadStats()
    testGetWeaponCatsNotNull() --> ohne

    prepChar()
    testGetWeaponCatsNotNull() --> ohne

LCOM = 3-ohne - 0-mit = 3
```

4 Determine cyclomatic complexity

Source of `de.harper hall.keeper.acid.weapons.GenericWeapon.startWeapon`. Long lines stripped, given that they are not relevant for cyclomatic complecity analysis.

```
01 private void startWeapon([...i]) throws SAXException {
02     String implementationClass = [...]
03
04     if ([...]) {
05         try {
06             tmpWeap = [...]
07         } catch (InstantiationException e) {
08             tmpWeap = null;
09             [...]
10             return;
11         } catch (IllegalAccessException e) {
12             tmpWeap = null;
13             [...]
14             return;
15         } catch (ClassNotFoundException e) {
16             tmpWeap = null;
17             [...]
18             return;
19         }
20     } else {
21         tmpWeap = new GenericWeapon();
22     }
23
24     if (tmpWeap instanceof DirectTableWeapon) {
25         DirectTableWeapon val = (DirectTableWeapon) tmpWeap;
26         String wTableName;
27
28         [...]
29
30         try {
31             val.setWTable([...]);
32         } catch (Exception e) {
33             [...]
34             tmpWeap = null;
35             return;
36         }
37
38         if (tmpWeap instanceof ParametrizedWeapon) {
39             ParametrizedWeapon pval = [...]
40
41             pval.setFumbleVal([...]);
42             pval.setCat([...]);
43         }
44     }
45 }
```

The cyclomatic complexity analysis results in a complexity of

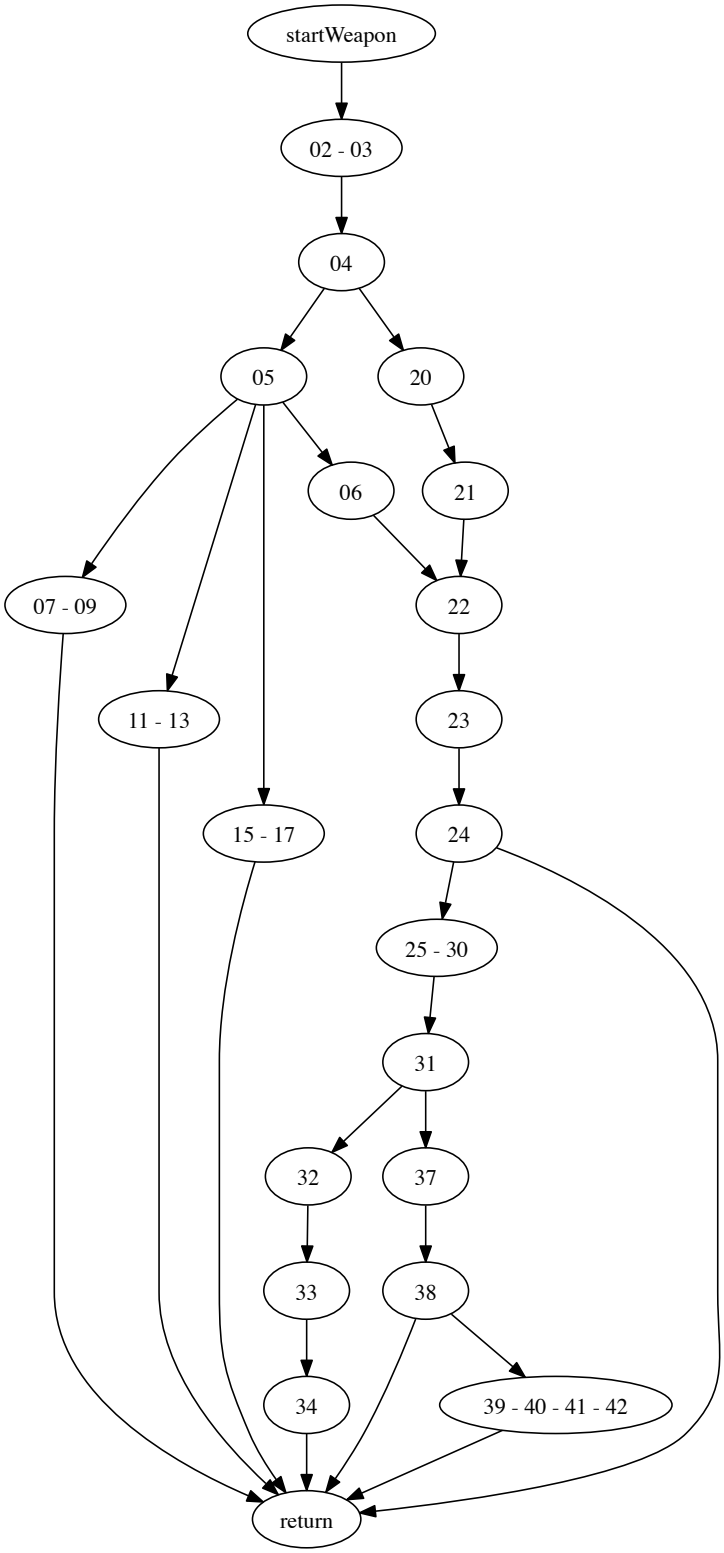
```
M = E - N + 2P = 28 - 22 + 2 * 1 = 8
```

where

```
E = the number of edges of the graph
N = the number of nodes of the graph
P = the number of connected components (exit nodes)
```

(Source: Wikipedia[1])

[1] http://en.wikipedia.org/wiki/Cyclomatic_complexity


startWeapon
02 - 03
04
05
20
06
21
07 - 09
22
11 - 13
23
15 - 17
24
25 - 30
31
32
37
33
38
34
39 - 40 - 41 - 42
return

5 Application complexity and comment quality

The overall complexity of the application is relatively high, given that it has many complex (cyclomatic number greater 5) and often very long (more than 40 lines) methods. The comment coverage is good, as far as we could determine it. However they could be more detailed, and they don't seem always be up to date.

- Overall complexity: 9
- Comment quality: 6

6 Playing with Sonar

- The most complex methods seems to be the constructor within `acid.WeaponTable` with a cyclomatic number of 31. The most complex class is `SkillDefinition`.

---

1. The code duplications detected by Sonar are not many. Apparently it only detects exact code duplications, which are mostly getters and setters in the case of Bookkeeper.
2. LCOM4 creates a graph which indicates what class methods access which class fields. If the resulting graph is not connected (that is, there is more than one component), LCOM4 indicates that the refactoring should be considered.
3. Sonar detects branch and line coverage (or statement coverage)
4. The complexity of the `GenericWeapon` class (as calculated by Sonar) is 52, with an average method complexity of 2.7.
5. Tests should be written covering many exception cases within `startWeapon`. The methods `startACMod()` and `getAllWeapons()` don't seem to tested at all.
6. The overall code complexity of the application is, as previously stated, relatively high (9 out of 10). There are 46 classes that are at least twice as complex as the average class, the most complex class being more than 14 times as complex.
7. Undocumented methods should definitely be documented. Sonar found 49 undocumented APIs. Also, the classes with the highest complexity, e.g. `SkillDefinition`, `KeeperCharacterBase` and `WeaponTable`, which are more than 8 times as complex as the average class, should be revised and possibly refactored.