GRUPPE9'S KODE STANDARD

Formål

Dette dokument skal beskrive Gruppe 9's anbefalede kode standard for at skrive Java programmer, hvor kode med "god stil" er beskrevet således:

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Efficient
- Completeness
- Correctness
- KISS (Keep-it-simple-stupid)

Naming

Vi benytter os af følge regler for namings.

Variable cases

```
1   PascalCase for types
2   camelCase for variables
3   UPPER_SNAKE for CONSTANTS
```

Figur1: Vores krav til variable cases

Variable naming

Et variable navn skal beskrive variablens formål. Hvis der tilføjes ekstra information såsom type i variablen, er det et tegn for et dårligt variable navn.

```
1   // Bad
2   Map<Integer, User> idToUserMap;
3   String valueString;
4
5   // Good
6   Map<Integer, User> usersById;
7   String value;
```

Figur2: Vores krav til variable navngivning

Class attribute naming

Følger de samme regler for variabler.

```
1   // Bad.
2   //   - Field names give little insight into what fields are used for.
3   class User {
4     private final int a;
5     private final String m;
6
7     ...
8   }
9
10  // Good.
11  class User {
12    private final int ageInYears;
13    private final String maidenName;
14
15    ...
16  }
```

Figur3: Dårlig og god navngivning for variabler

Include units in variable names

Der skal inkluderes units i variable names, så det gør forståelsen for forskellen mellem variabler og læsbarheden høj.

```
// Bad
long pollInterval;
int fileSize;

// Good
long pollIntervalMs;
int fileSizeGb;
```

Figue4: Dårlig og god måde at vise units for en variable

Space pad operators and equals

Der skal være klar visuel separation af operationer, så det giver en bedre læsbarhed.

```
// Bad
int foo=a+b+1;

// Good
int foo = a + b + 1;
```

Figur5: Dårligt og godt eksempel på separation

Indent style

Vi gør brug af ”one true brace style aka 1TBS”. Dvs. at 1TBS har sin åbne brace på starten af linjen.

```
// Bad
public void memorize()
{

}

// Good
public void memorize() {

}
```

Figur5: Dårlig og god brug af brackets

Kommentar

Vi skriver kommentar til de metoder vi mener er komplekse, dvs. kode som vi ikke vil forstå i morgen, om 1 uge, eller om 1 år. Det vil ikke sige getters og setters i en klasse.

```
1   // Bad
2   // This code returns the monkeys name
3   public String getMonkeyName(Monkey monkey) {
4
5   }
6
7   // Good
8   /**
9    * params Monkey object
10   * returns monkey name
11   * kommentar
12   */
13  public String getMonkeyName(Monkey monkey) {
14
15  }
```

Firgur6: Dårlig og god kommentering af funktioner

Funktioner

Funktioner må ikke være mere end 10 linjer, hvis dette overskrives, skal der refactors så man får flere funktioner. Linjer må ikke gå udover midterskærmen i NetBeans.

Returns

Der skal altid returnes tidligt i koden, hvis nødvendigt. Der må ikke returnes midt i en kode block.

Kode reviews

Der lave pair reviews på alt kode, derfor laver vi pull requests som har minium krav at en anden skal tjekke og godkende koden i guthub, før den kan blive merged ind i master branchen.

Branching

Alt koden der ligger i master, skal være deployable. Derfor tester hver udvikler deres kode før de merger den til master og tjekkes i kode review.

Performance

Vi har ikke lagt vægt i at lave performance optimering, da dette er et relativt lille system og ikke skal kunne skalere til flere 10000 brugere.

Implementerings praksisser

Vi gør brug af defensive programming, hvor vi laver tjek af conditions før der gøres brug af dem. Dvs.

```
1    public void dropItem(Command command) {
2        if (!command.hasSecondWord()) return;
3        String itemName = command.getSecondWord();
4
5        ...
6    }
```

Figur7: Eksempel på defensive programming

Linter

Vi vil forsøge at implementere en linter/code analyser i NetBeans for at forholde vores kode standarder.