Software Reengineering: **Duplicate Code**

Henrique Rocha

Universiteit Antwerpen

Introduction

- Duplicated code is a phenomena that could be as old as programming itself.
  - Some old programming languages did not have support for reuse of code (or easy reuse) as we have today.
- It is easy to understand the appeal of duplicating code:
  - Easy to do it in every programming language
  - Fast development

What is the Problem with Duplicate Code

"Fowler and Beck have ranked duplicated code as the first of the top ten code smells indicating the need to refactor a piece of software [FBB+99]." (OORP, p.223).

When you make duplicate code you are taking a **loan**…

…which you will have to 'pay' later.

These types of design decisions considered "loans" are more used referred as **Technical Debt**.

Disadvantages of Duplicate Code

- Makes Maintenance More Difficult
  - Copying Bugs
  - Managing maintenance across duplicates
- Hinders the Understandability
  - Code spread through your system
- Increases compile time
  - Increased LOC

Definitions

- A duplicate code is called a **Code Clone**.
- A set of equivalent clones is called a **Clone Class**.

- Clones are classified into four types.
  - Type 1 : Identical clones
  - Type 2: Structurally Identical clones
  - Type 3: Structurally Identical clones with Gaps
  - Type 4: Semantic clones

Type 1 Clone – Example

File Name: A Start Line: 1 End Line: 5

```
1  void swap(int &a,int &b){
2      int temp = a;
3      a = b;
4      b = temp;
5  }
```

File Name: B Start Line: 1 End Line: 9

```
1  void swap(int &a,
2            int &b) {
3
4  int temp = a;  // temporary variable used to swap the value
5
6  a = b;
7  b = temp;
8
9 }
```

Type 2 Clone – Example

File Name: A Start Line: 1 End Line: 5

```
1  void swap(int &a, int &b){
2      int temp = a;
3      a = b;
4      b = temp;
5  }
```

File Name: B Start Line: 1 End Line: 9

```
1  void swap(int &x,
2            int &y) {
3
4  int z = x;  // temporary variable used to swap the value
5
6  x = y;
7  y = z;
8
9 }
```

Type 3 Clone – Example

File Name: A Start Line: 1 End Line: 5

```
1  void swap(int &a, int &b){
2      int temp = a;
3      a = b;
4      b = temp;
5  }
```

File Name: B Start Line: 1 End Line: 9

```
1  void swap(int &x,
2            int &y) {
3
4  int z = x;  // temporary variable used to swap the value
5
6  x = y;
7  y = z;
8  cout<<"Current values are "<<x<<","<<y;
9 }
```

Type 4 Clone – Example

File Name: A Start Line: 11 End Line: 18

```
11 int sum(int a){
12   int result =0;
13   for(int i=1;i<=a;i++){
14      result = result + i;
15   }
16   return result;
17
18 }
```

File Name: B Start Line: 25 End Line: 34

```
25  int addition(int limit) {
26     int sum = 0;
27     int counter = 1;
28     while(true){
29       sum = sum + counter;
30       if((counter=counter+1)>limit)
31       break;
32     }
33     return sum;
34  }
```

Clone Detection Techniques

Older, Studied Extensively — Newer, Bleeding Edge

Less Complex — More Complex

Syntactic Understanding — Semantic Understanding

Code Cloning and Reengineering

- For Reengineering we want to improve the overall quality of the software
  - And increase maintainability, understandability, …
- Usually, 5% to 10% of the code in large systems is duplicated

When Clones are Justified?

- Sometimes Clones can be a reasonable design decision
  - As well as leaving them alone in the system
- Some cases where that is justified:
  - Constraints inherent to the program-language
  - Less complex than making abstractions
  - Prevent the system from becoming unstable

Clones in the Project

- For the Intermediate Report, you need to run a clone detection tool and reason on which clones you will remove (and which one you wont)
  - Not every clone needs to be removed, only those affecting the parts necessary for your project.
  - Check the pattern “If it Ain’t Broke, Don’t Fixed” (OORP, p.35)

- For the Final Report, it is important to highlight the reasons and design decisions on which clones you removed (and which ones you chose to ignore).