VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Advanced Programming (CO2039)

Report (Semester 202, Duration: 01 weeks)

OOP vs FP

Advisor: Mr. Lê Lam Sơn

Student Name: Nguyễn Hoàng
Student ID: 1952255

HO CHI MINH CITY, AUGUST 2021

1 OOP and FP in baking a pizza

OOP makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

What? Alright that sounds a bit rough, let's rephrase this a bit. OOP aims to model the world in self-contained entities, and affects change by modifying the state of itself or other entities. FP on the other hand aims to not modify the original data, but rather creates new data given some existing data.

To demonstrate this, we will try to make a pizza. With OOP, a big box or object with all the materials to create a pizza is available, and the helper methods will slowly transform them into a complete pizza. FP will take a different approach, as materials are given to each stage/step/activity in order to be used in the next activity until the final product is achieved.

We will try to describe this pizza making progress programmatically using C++ and Haskell.

Let's start with a complete C++ program

```cpp
#include <iostream>
#include <string>

class Pastry
{
public:
  virtual void bake_me_baby()
  {
    prepare_dough();
    add_sauce();
    add_toppings();
    bake();
  }

protected:
  virtual void prepare_dough() = 0;
  virtual void add_sauce() = 0;
  virtual void add_toppings() = 0;
  virtual void bake() = 0;
};

class Pizza : public Pastry
{
protected:
  int time = 0;
  std::string state = "Raw";

protected:
  void prepare_dough()
  {
    if (time != 0)
      return;
    time = 1;
```

```
34      state = "Prepared dough";
35    }
36    void add_sauce()
37    {
38      if (time != 1)
39        return;
40      time = 2;
41      state = "Added sauce";
42    }
43    void add_toppings()
44    {
45      if (time != 2)
46        return;
47      time = 3;
48      state = "Added toppings";
49    }
50    void bake()
51    {
52      if (time != 3)
53        return;
54      time = 4;
55      state = "Baked the hell out of this";
56    }
57
58  public:
59    void bake_me_baby()
60    {
61      prepare_dough();
62      std::cout << time << " " << state << std::endl;
63      add_sauce();
64      std::cout << time << " " << state << std::endl;
65      add_toppings();
66      std::cout << time << " " << state << std::endl;
67      bake();
68      std::cout << time << " " << state << std::endl;
69    }
70  };
71
72  int main(int argc, char **argv)
73  {
74    Pastry *pizza = new Pizza();
75    pizza->bake_me_baby();
76    delete pizza;
77    return 0;
78  }
```

Output of this program

```
1 Prepared dough
2 Added sauce
3 Added toppings
4 Baked the hell out of this
```

Nice! Let's do this again, but with Haskell

```haskell
module Main (main) where

data Pastry = Pizza {time :: Int, state :: String} deriving (Show)

prepareDough :: Pastry -> Pastry
prepareDough pizza@(Pizza t _)
  | t /= 0 = pizza
  | otherwise = pizza {time = 1, state = "Prepared dough"}

addSauce :: Pastry -> Pastry
addSauce pizza@(Pizza t _)
  | t /= 1 = pizza
  | otherwise = pizza {time = 2, state = "Added sauce"}

addToppings :: Pastry -> Pastry
addToppings pizza@(Pizza t _)
  | t /= 2 = pizza
  | otherwise = pizza {time = 3, state = "Added toppings"}

bake :: Pastry -> Pastry
bake pizza@(Pizza t _)
  | t /= 3 = pizza
  | otherwise = pizza {time = 4, state = "Baked the hell out of
  this"}

bakeMeBaby :: Pastry -> IO ()
bakeMeBaby pizza = do
  let pizza1 = prepareDough pizza
  print pizza1
  let pizza2 = addSauce pizza1
  print pizza2
  let pizza3 = addToppings pizza2
  print pizza3
  let pizza4 = bake pizza3
  print pizza4

main :: IO ()
main = do
  let pizza = Pizza 0 "Raw"
  bakeMeBaby pizza
```

Output of this program

```
Pizza {time = 1, state = "Prepared dough"}
Pizza {time = 2, state = "Added sauce"}
Pizza {time = 3, state = "Added toppings"}
Pizza {time = 4, state = "Baked the hell out of this"}
```

2 Conclusion

With the pizza making out of the way, there are definitely some things noticeable between the two approaches. Obviously, the procedure of the process does not change, but the way the materials or the pizza, otherwise known as the data, are handled and processed is different.

	OOP	FP
Pros	OOP objects contains both the data (attributes) and things that it can do (methods)	FP decouples the data from the functions
	Any changes that are applied is reflected on the object itself	Data in FP is not intended to change, if it does, new data is just created
	OOP organize everything into hierarchies of abstract objects	FP is natural for the human brain, as in, our thought process is centered around "doing" things
Cons	OOP objects are abstract and potentially complex, wasting a lot of time in abstractions and design patterns instead of solving the problem	
	OOP encourages sharing of mutable state and introduces additional complexity with its numerous design pattern, thus making common development practices, like refactoring and testing, needlessly hard	
	Concurrency is basically impossible because the output of a method depends on the state of the object, unless we make the object immutable, which is FP	

Easter egg

Congratulations! You have actually read my report. This section serves no more than empty space that I type when I can't get problems out of my mind.