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()
  {
    time = 1;
    state = "Prepared dough";
  }
```

```
  void add_sauce()
  {
    time = 2;
    state = "Added sauce";
  }
  void add_toppings()
  {
    time = 3;
    state = "Added toppings";
  }
  void bake()
  {
    time = 4;
    state = "Baked the hell out of this";
  }

public:
  void bake_me_baby()
  {
    prepare_dough();
    std::cout << state << std::endl;
    add_sauce();
    std::cout << state << std::endl;
    add_toppings();
    std::cout << state << std::endl;
    bake();
    std::cout << state << std::endl;
  }
};

int main(int argc, char **argv)
{
  Pastry *pizza = new Pizza();
  pizza->bake_me_baby();
  delete pizza;
  return 0;
}
```

Output of this program

```
>>> g++ -g main.cpp -o main && ./main
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

```
module Main (main) where

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

```haskell

prepareDough :: Pizza -> Pizza
prepareDough pizza = pizza {time = 1, state = "Prepared dough"}

addSauce :: Pizza -> Pizza
addSauce pizza = pizza {time = 2, state = "Added sauce"}

addToppings :: Pizza -> Pizza
addToppings pizza = pizza {time = 3, state = "Added toppings"}

bake :: Pizza -> Pizza
bake pizza = pizza {time = 4, state = "Baked the hell out of
this"}

bakeMeBaby :: Pizza -> 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

```
>>> ghc -g main.hs -o main && ./main
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking main ...
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 Comparing the pizza-making methods