Creational Design Patterns

Part 4

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Creational Patterns: Builder

- Suppose we are building a new web site for Pizza Pizza

- We have to support two types of pizza:
  - Pre-defined pizzas: Pepperoni and Cheese, Hawaiian, Deluxe, etc.
  - Custom pizzas

Creational Patterns: Builder

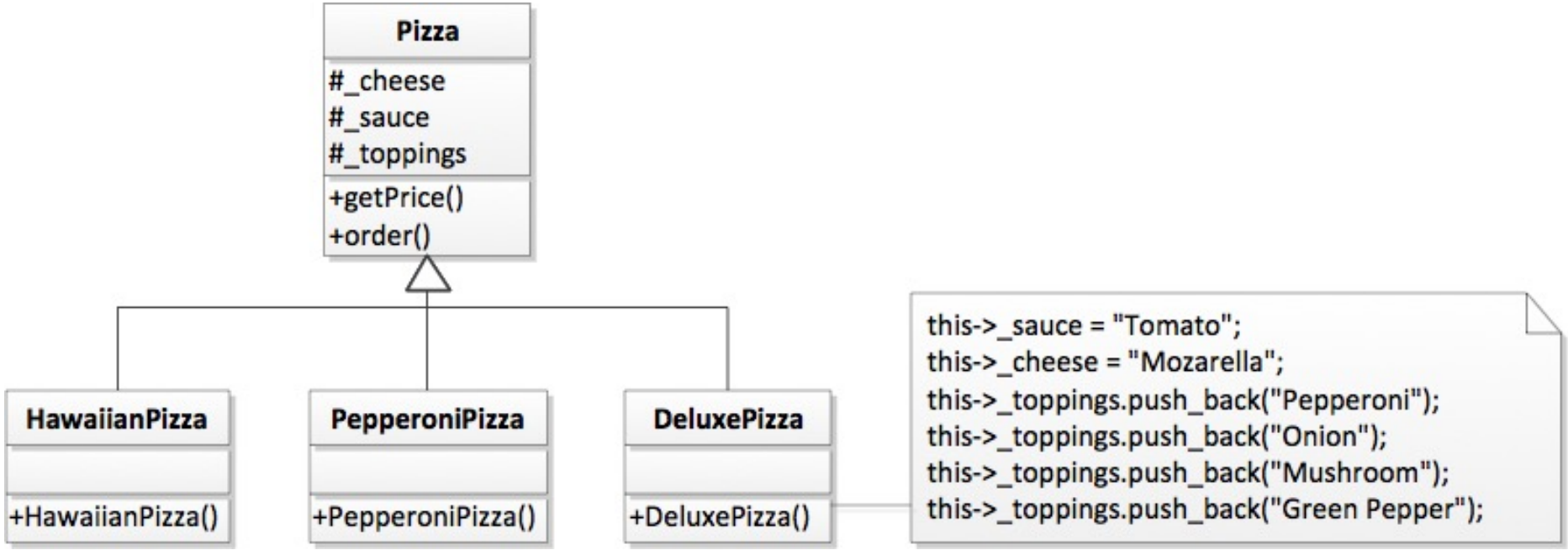
We might have the following code in various places throughout our application:

```
// Build a Hawaiian pizza
Pizza *pizza = new Pizza(12);  // 12" pizza
pizza->addTopping("Pineapple");
pizza->addTopping("Ham");

// ...

// Build a Deluxe pizza
Pizza *pizza = new Pizza(8);
pizza->addTopping("Pepperoni");
pizza->addTopping("Mushroom");
pizza->addTopping("Green Peppers");
pizza->addTopping("Onions");
```

Creational Patterns: Builder

- This can be cumbersome and error-prone
  - We might forget to add green peppers to a Deluxe pizza in one part of our application
- It would be ideal to encapsulate this creation process
- One possible solution involves sub-classing …

Creational Patterns: Builder



Creational Patterns: Builder

- Sub-classing seems like overkill for this application:
  - Our subclasses do not add new state or behaviour
  - Instead, they merely create different representations of the same thing: a pizza!

- How can we create these different representations without adding new sub-classes?

Creational Patterns: Builder

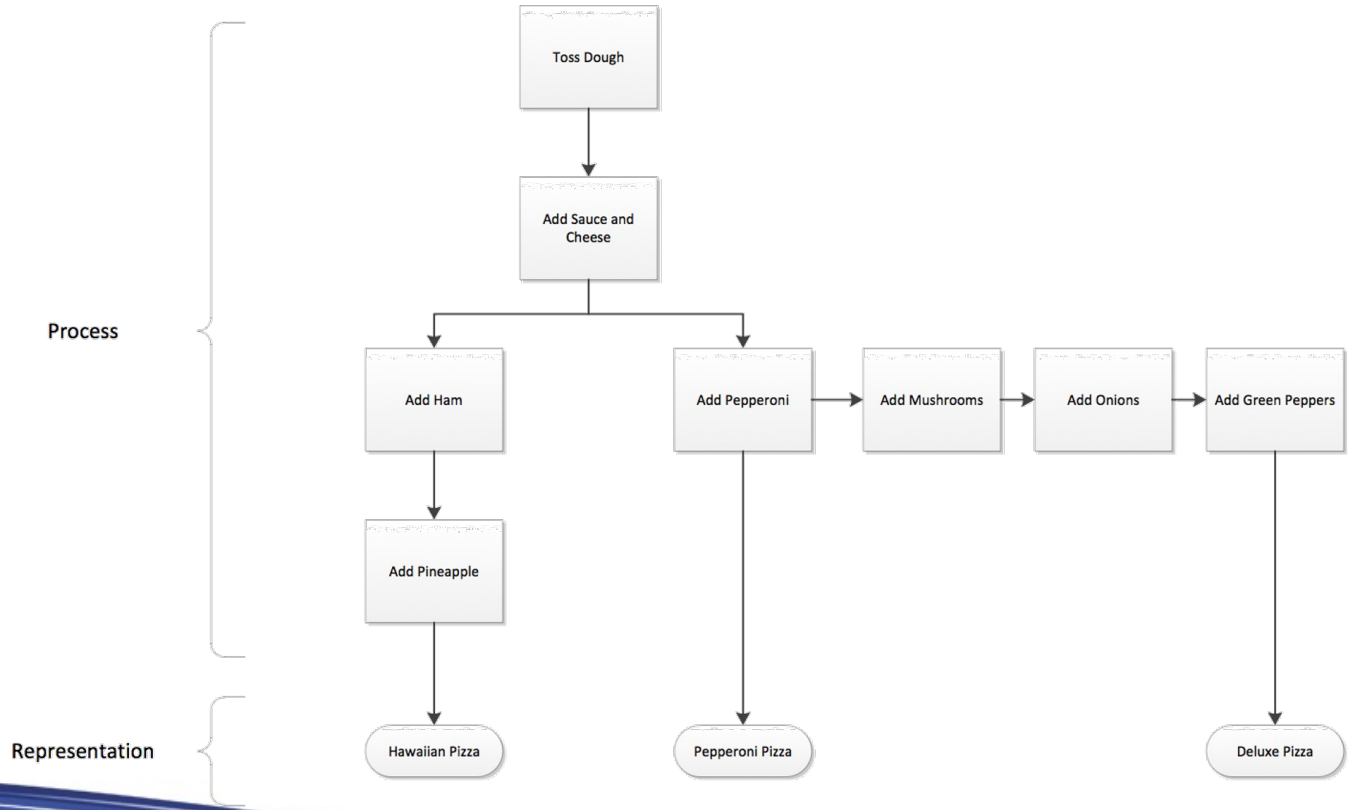
Design Pattern:
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

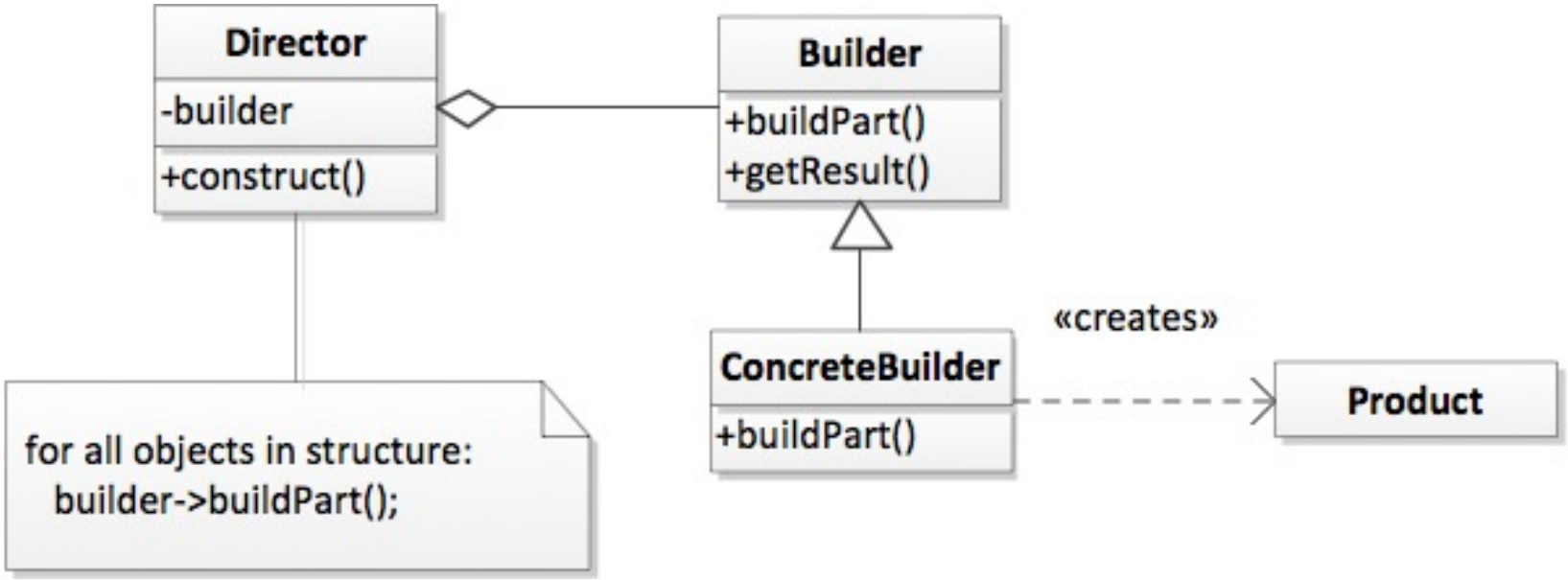
Creational Patterns: Builder

- Applicability:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - The construction process must allow different representations for the object that's constructed

Creational Patterns: Builder



Creational Patterns: Builder

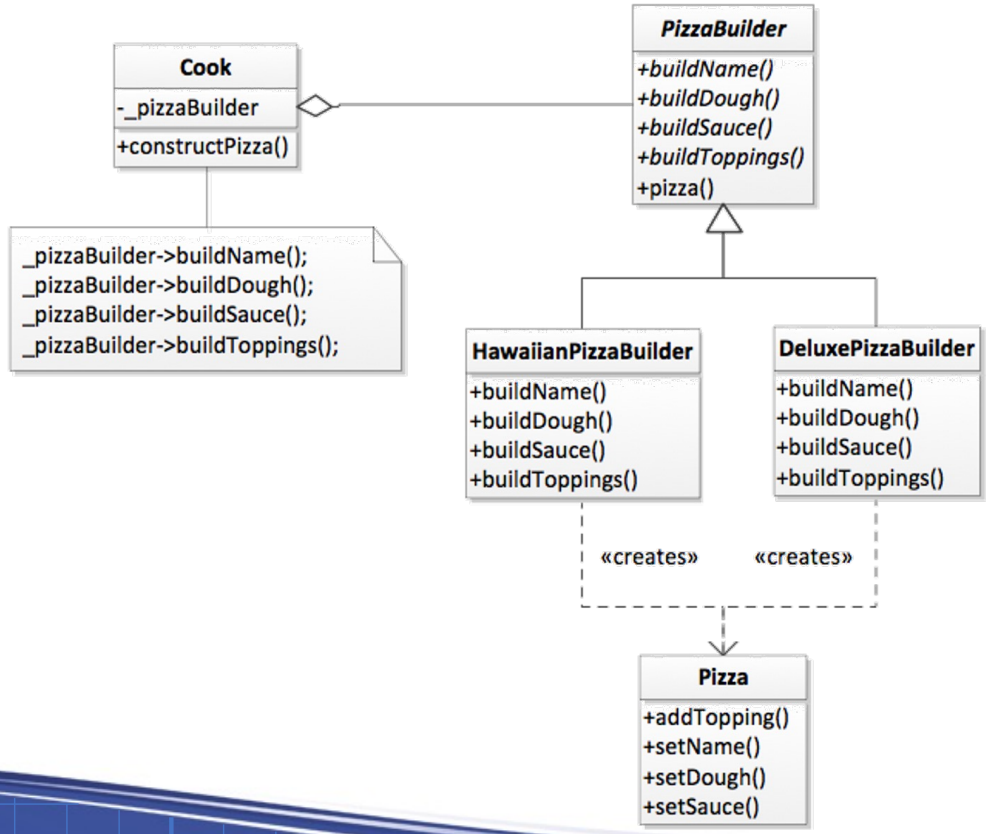


Creational Patterns: Builder

Classes:

- Director
  - Responsible for the sequence of build operations
- Builder
  - Abstract interface for creating products
- Concrete Builder
  - Implements construction and assembly of parts
- Product
  - Object that will be created by Concrete Builder

Creational Patterns: Builder



Creational Patterns: Builder

PizzaBuilder.h

```
// Abstract Builder

class PizzaBuilder
{
   public:
      const Pizza& pizza()
      {
         return _pizza;
      }
      virtual void buildName() = 0;
      virtual void buildDough() = 0;
      virtual void buildSauce() = 0;
      virtual void buildToppings() = 0;

   protected:
    Pizza _pizza;
};
```

Creational Patterns: Builder

HawaiianPizzaBuilder.cpp

```
void HawaiianPizzaBuilder::buildName()
{
   _pizza.setName("Hawaiian");
}

void HawaiianPizzaBuilder::buildDough()
{
   _pizza.setDough("Regular");
}

void HawaiianPizzaBuilder::buildSauce()
{
   _pizza.setSauce("Mild");
}

void HawaiianPizzaBuilder::buildToppings()
{
   _pizza.addTopping("Ham");
   _pizza.addTopping("Pineapple");
}
```

Creational Patterns: Builder

DeluxePizzaBuilder.cpp

```
void DeluxePizzaBuilder::buildName()
{
   _pizza.setName("Deluxe");
}
void DeluxePizzaBuilder::buildDough()
{
   _pizza.setDough("Thick");
}
void DeluxePizzaBuilder::buildSauce()
{
   _pizza.setSauce("Mild");
}
void DeluxePizzaBuilder::buildToppings()
{
   _pizza.addTopping("Pepperoni");
   _pizza.addTopping("Mushrooms");
   _pizza.addTopping("Onions");
   _pizza.addTopping("Green Peppers");
}
```

Creational Patterns: Builder

Cook.cpp

```
Cook::Cook() : _pizzaBuilder(NULL)
{
}
Cook::~Cook()
{
   if (_pizzaBuilder)
      delete _pizzaBuilder;
}
void Cook::setPizzaBuilder(PizzaBuilder* pizzaBuilder)
{
   if (_pizzaBuilder)
      delete _pizzaBuilder;

      _pizzaBuilder = pizzaBuilder;
}
const Pizza& Cook::getPizza()
{
   return _pizzaBuilder->pizza();
}
void Cook::constructPizza()
{
   _pizzaBuilder->buildName();
   _pizzaBuilder->buildDough();
   _pizzaBuilder->buildSauce();
   _pizzaBuilder->buildToppings();
}
```

Creational Patterns: Builder

main.cpp

```
int main()
{

   Cook cook;
   cook.setPizzaBuilder(new HawaiianPizzaBuilder);
   cook.constructPizza();

   Pizza hawaiian = cook.getPizza();
   cout << hawaiian << endl;

   cook.setPizzaBuilder(new DeluxePizzaBuilder);
   cook.constructPizza();

   Pizza deluxe = cook.getPizza();
   cout << deluxe << endl;
}
```

Creational Patterns: Builder

- Consequences:
  - Lets you vary a product's internal representation
  - Isolates code for construction and representation
  - Gives you finer control over the construction process

Creational Patterns: Builder

- Builder vs. Abstract Factory
  - Abstract Factory
    - Deals with families of related objects
    - Available immediately
  - Builder
    - Creates one, complex product, usually made up of different parts
    - Available via `getResult()`