COMP 2511
Object Oriented Design & Programming

Week 08

Last week,

Composite, Decorator, Observer Patterns

Model View Controller Software Architecture

OO design principles

- Open Closed Principle
- Strive for loosely coupled objects

This week

Design Patterns

- Factory, Builder Patterns
- Visitor Pattern

Factory Patterns

Factory Method Pattern
And
Abstract Factory Pattern

Patterns so far

- Behavioural Patterns
  - Strategy, State, Observer, Template Method
- Structural Patterns
  - Composite, Decorator
- Creational Patterns
  - Factory Method, Abstract Factory

Creational Patterns

- Abstract the object instantiation process, by hiding how objects are created
- Class Creational patterns use inheritance to decide the object to be instantiated
  - Factory Method Pattern
- Object Creational patterns delegate the instantiation to another object
  - Abstract Factory

Joe's Security Application

A security application:

- Allows a user to supply some text and a filename, which is then encrypted and written to disk
- Different encryption algorithms may be needed based on different scenarios

Think “concrete” when you see new()

- Remember our design principle:

  “ Program to an interface, not to an implementation ”

```
EncryptionAlgorithm al = new SHA256EncryptionAlgorithm();
```

We want to use interfaces to keep code flexible

But, we have to create an instance of a concrete class

- Several concrete classes (`SHA256EncryptionAlgorithm`, `SHA512EncryptionAlgorithm`) and the decision which concrete class to instantiate is made at run-time, depending on a set of conditions

Run-time instantiation of the concrete class

We are passing in the type of encryption technique to method encryptToDisk

```
public EncryptAlgorithm encryptToDisk(
String text, String fileName, String type) {

EncryptAlgorithm algorithm;

if (type.equals("sha256")) {
   algorithm = new SHA256EncryptAlgorithm();
}
else if (type.equals("sha512")) {
   algorithm = new SHA512EncryptAlgorithm();
}

return algorithm;
}
```

Based on the type of algorithm, we instantiate the concrete class and assign it to the algorithm instance variable. Each concrete class extends the abstract class EncryptAlgorithm

Responding to more security threats…

- Add more complex encryption techniques
- Remove easy hack techniques

How does the software change?

What code varies, as algorithm options changes?

Which part of code varies?

Remove unsecure techniques

```
public EncryptAlgorithm encryptToDisk(...
       String type) {
EncryptAlgorithm algorithm;

if (type.equals("sha256")) {
   algorithm = new SHA256EncryptAlgorithm();
}
else if (type.equals("sha512")) {
   algorithm = new SHA512EncryptAlgorithm();
}
else if (type.equals("md5")) {
   algorithm = new MD5Algorithm();
}

return algorithm;
}
```

This is the code that varies – the process of instantiation of the encryption algorithm

- Remove algorithms, change code
- Add algorithms, change code

Ongoing problem, as algorithms change

This design is clearly **NOT closed to modification**

Code error-prone, hard to maintain – **Design Smell of Fragility**

Encapsulate object creation

```
public EncryptAlgorithm
encryptToDisk(... String type) {
  EncryptAlgorithm algorithm;

  /* Pull out the object creation
  code into a new class called
  Factory, which handles the
  creation of objects */

  return algorithm;
  }
```

EncryptAlgorithmFactory

```
if (type.equals("sha256")) {
  algorithm = new SHA256EncryptAlgorithm();
}
else if (type.equals("sha512")) {
  algorithm = new SHA512EncryptAlgorithm();
}
else if (type.equals("md5")) {
   algorithm = new MD5EncryptAlgorithm();
}
```

But haven't we just pushed off the problem to a new class ?

Yes, but..

- Many clients may use the factory, and by encapsulating the algorithm creation in one place, only one place needs to change when modifications are made
- However, the factory class does not conform to OCP

Using the EncryptAlgorithmFactory

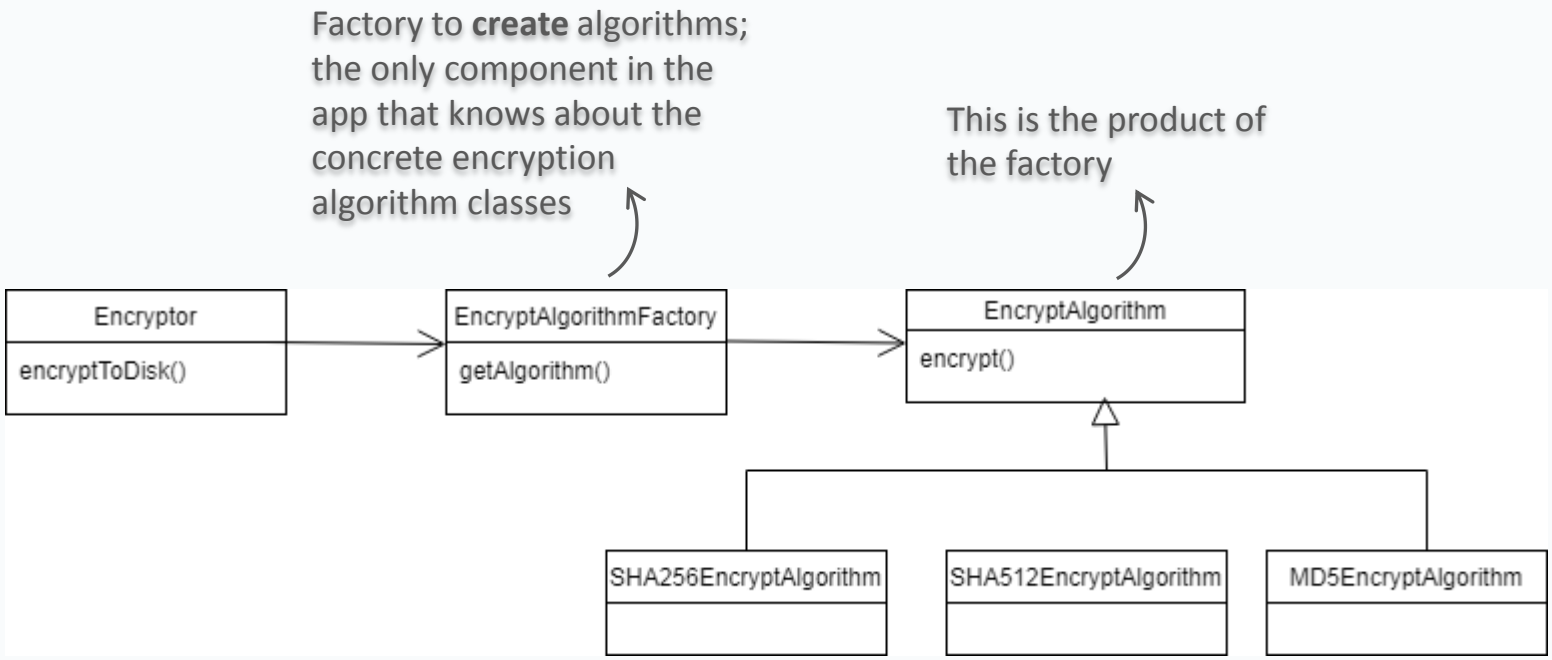
```
public class Encryptor{

 private  EncryptAlgorithm factory;

 public Encryptor(EncryptAlgorithmFactory
                  factory)
 {
       this.factory = factory;
 }

 public void encrpytToDisk(
   String text, String fileName, String type)
 {

   EncryptAlgorithm algorithm;
   algorithm = factory.createAlgorithm(type)

   // other methods
   algorithm.encrypt(text);
   // write to disk;
 }
```

Encryptor gets the factory passed to it in the constructor

getAlgorithm() method uses the factory to create its algorithms by simply passing on the type of the encryption technique needed

We have replaced **new** with a **create** method and **no instantiation of concrete classes** in Encryptor

The class diagram for our encryptor



Simple Factory Class

- Our previous design is often referred to as the “Simple Factory”
- The Simple Factory Class (`EncryptAlgorithmFactory`) tries to abstract the creation details of the product from the caller
- However, note, the Simple Factory is NOT a design pattern
- It is NOT the same as Factory Method Design Pattern!!

Factory Method Pattern

- Commonly misunderstood to mean any method whose sole purpose is to construct an object and return this created object (like the Simple Factory)

- The Factory Method Design Pattern uses inheritance to solve the problem of creating objects without specifying their exact object classes

Meet the Factory Method Pattern

Motivation

- A class (*creator*) cannot anticipate the class of objects it must create (*product*)
- Subclasses of the creator type need to create different kinds of product objects

Intent

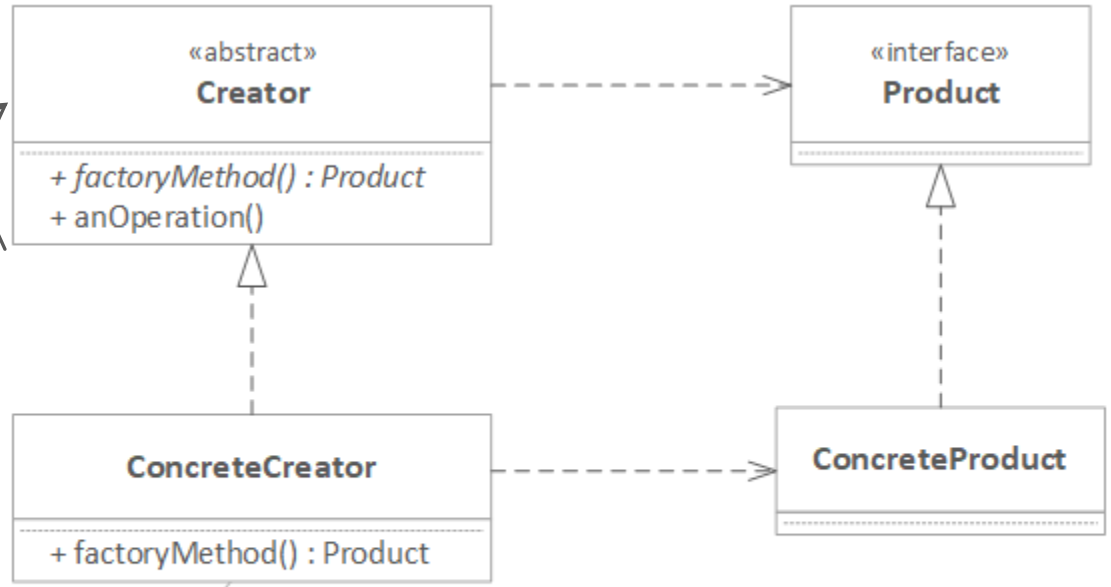
- The Factory Method Pattern is a creational design pattern that that provides an interface for creating objects in superclass, but allow sub-classes to alter the type of objects that will be created
- The Factory Method Pattern lets a class defer instantiation to subclasses

Factory Method Pattern: UML class diagram

- Create two sets of **`interfaces`** (1) for the **`Product`** that defines the object to be created (2) the **`Creator`** interface that constitutes the object that will instantiate the **`Product`**
- The **`Creator`** interface includes one or more factory methods as well as other methods
- A set of **`ConcreteCreator`** classes are created that return desired **`ConcreteProduct`** instances that implement the **`Product`** interface

A class that contains the implementations for all of the methods to manipulate products except for the factory method

The abstract factory method is what all concrete classes must implements

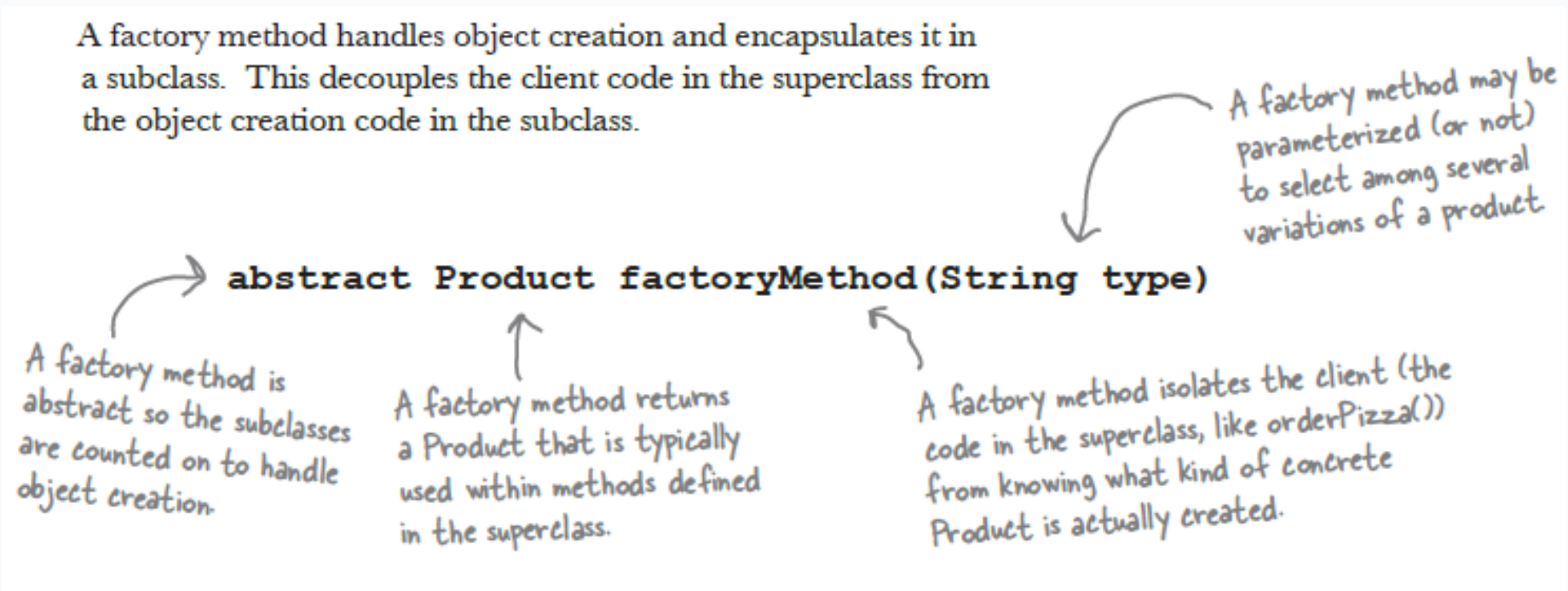


The concrete creator creates one or more concrete products. It is the only class that has knowledge of how to create the concrete Product classes

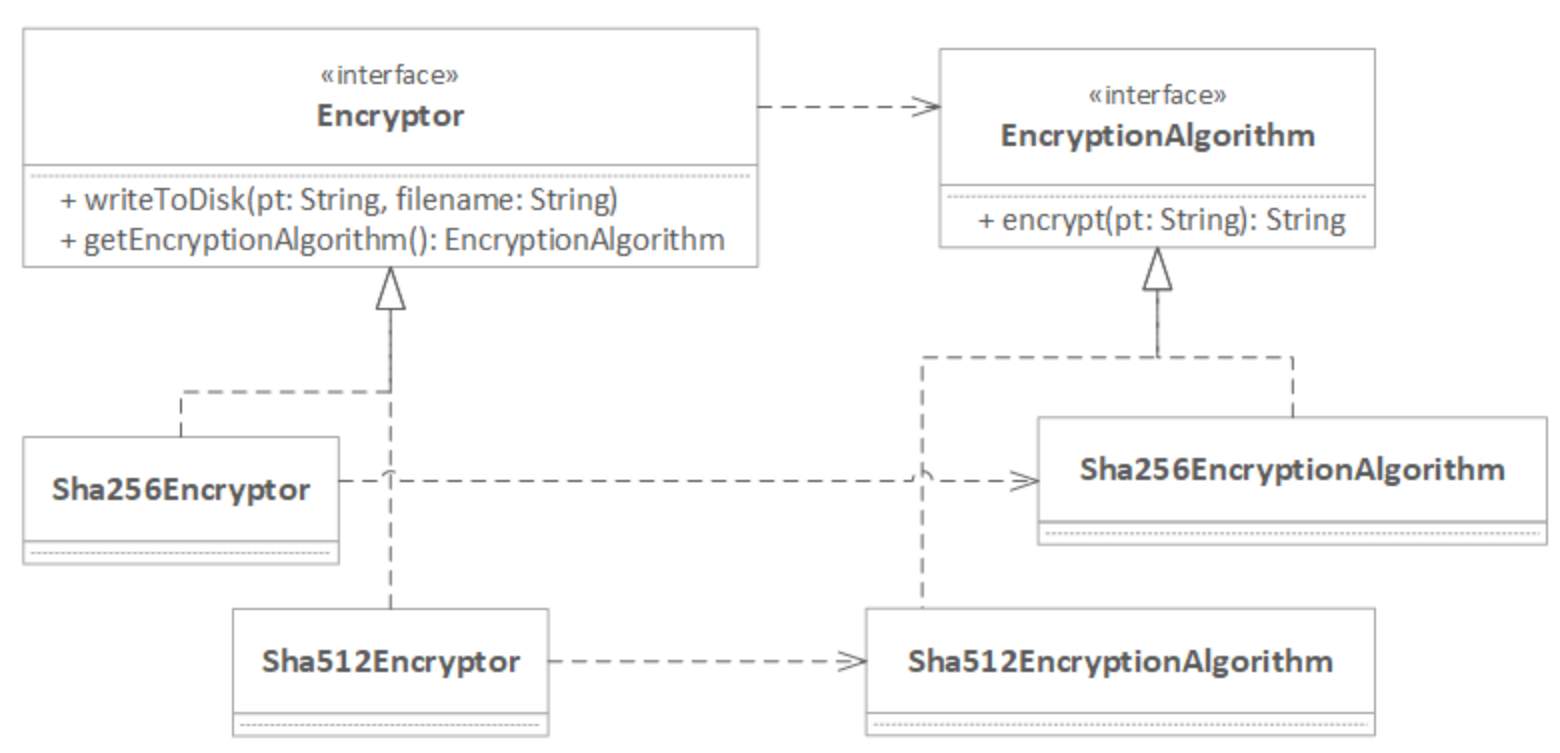
Factory Method Pattern: Abstract Method

The **`Creator`** interface includes one or more factory methods as well as other methods

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

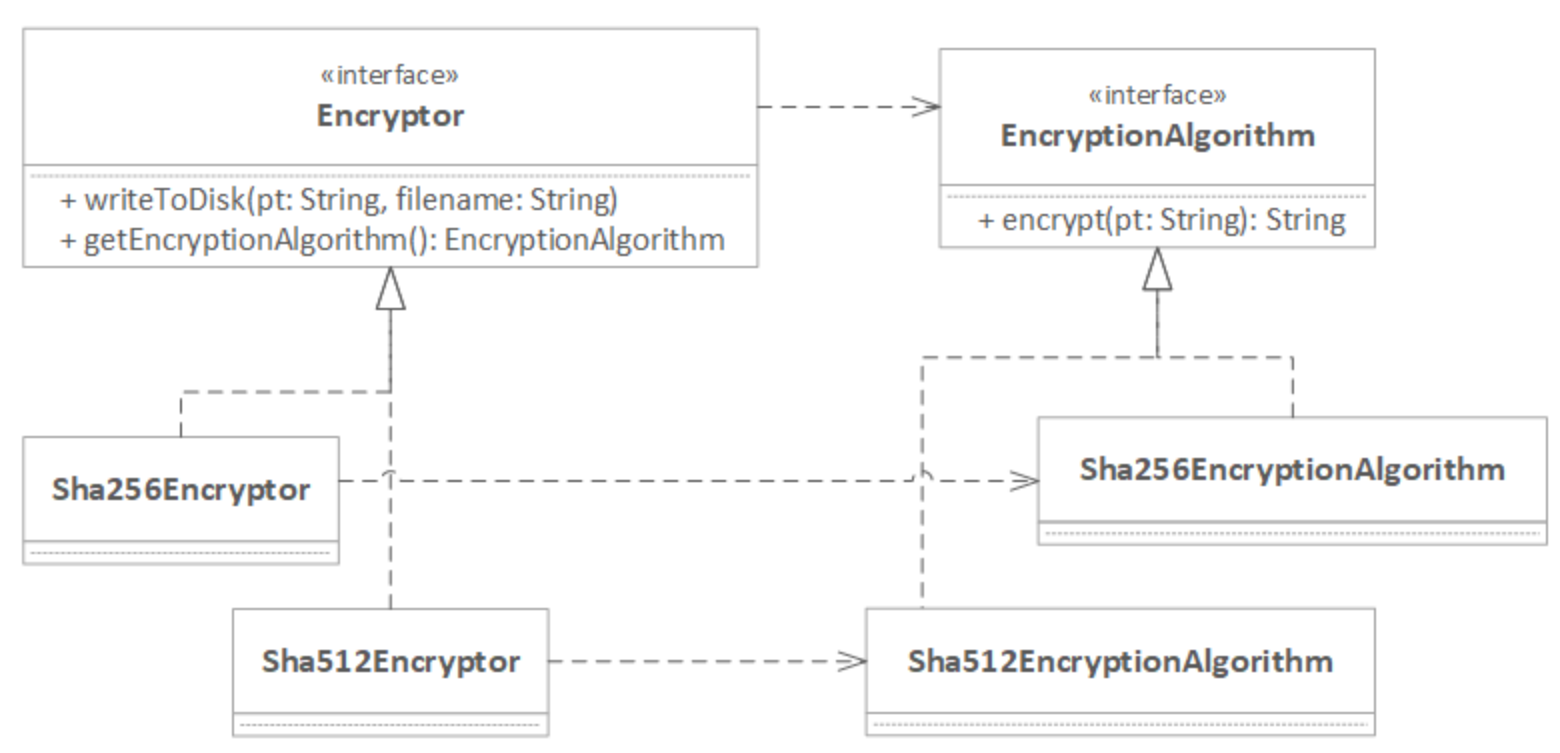


Implementing our example with the Factory Method Pattern



Lecture Demo: Encryptor example…

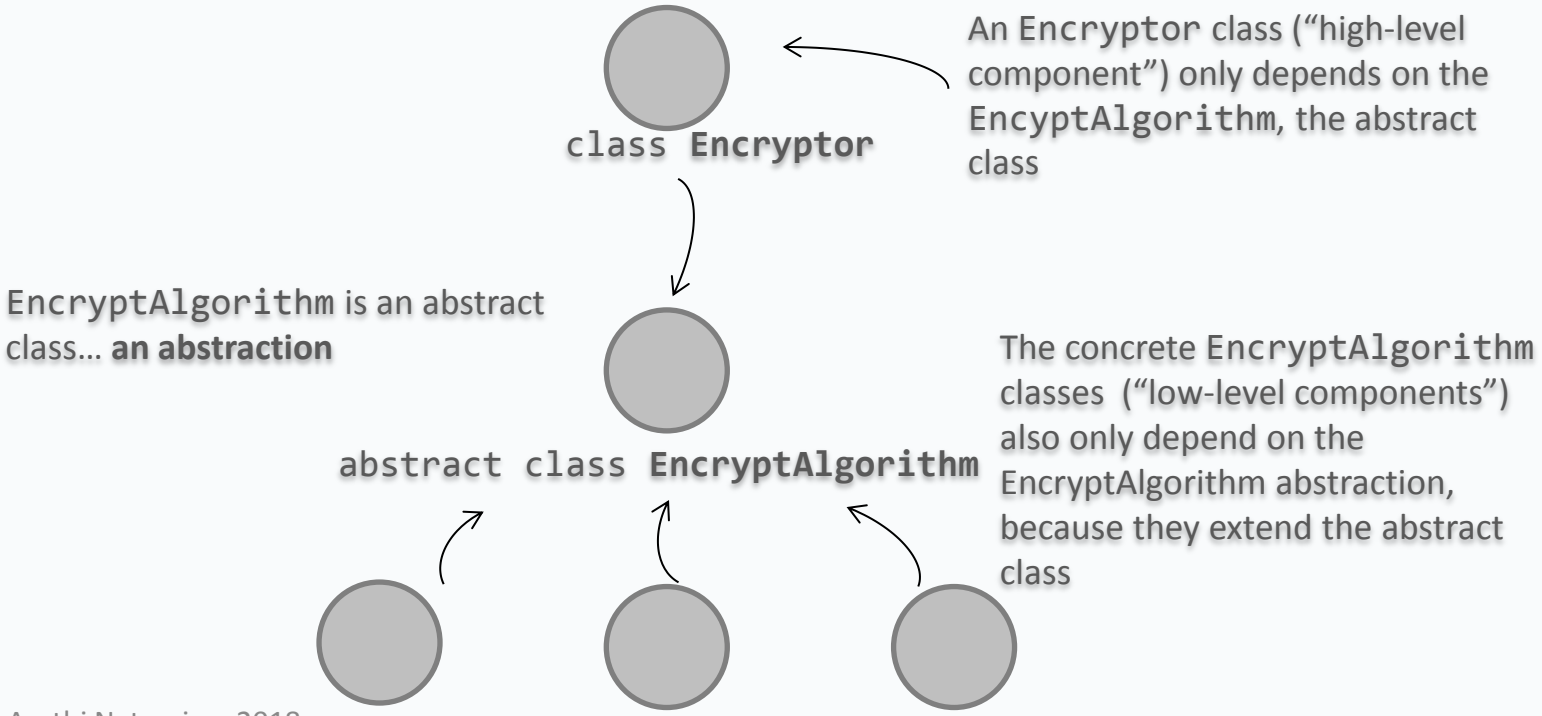
Implementing our example with the Factory Method Pattern



Lecture Demo: Encryptor example…

Design Principle #8: Dependency Injection Principle

- Depend upon abstractions. Do not depend on create classes
- A high-level module should not depend on "low-level" modules, but both should depend on abstractions



Factory Method in Collections Framework

Name in Design Pattern	Actual Name
Creator	Collection
ConcreteCreator	A sub class of collection (e.g., ArrayList
factorymethod()	Iterator()
Product	Iterator
ConcreteProduct	A sub class of Iterator (e.g., ListIterator

Factory Method Pattern Pros and Cons

- Follows the Open Closed Principle
- Avoids tight coupling between concrete products and the client that uses these products
- Localises all creational code to one place
- Extensible, allows news products to be created

However,

- Requires extra sub-classes
- May include a dense degree of code to support the features provided by the pattern implementation
- If the factory method needs to be able to create multiple kinds of products, then the factory method needs to take a parameter (possibly used in an if-else loop …. Violation of OCP)

The Abstract Factory Method Pattern

Motivation

- Need a way to produce families of related objects without specifying their concrete classes
  - a family of related furniture: chair, sofa, coffee table
  - and several variants of this family: Victorian Style, Georgian, ArtDeco
    - Here, need a way to create individual furniture objects, so that they match other objects of the same family (e.g., adding a dining table in the Victorian Style, Georgian etc)
    - And you do not want to change existing code, when new products or families of products are added

Intent

- The Abstract Factory Method Pattern is a creational design pattern that that provides an interface for creating families of related or dependent objects without specifying their classes

The Abstract Factory Method Pattern

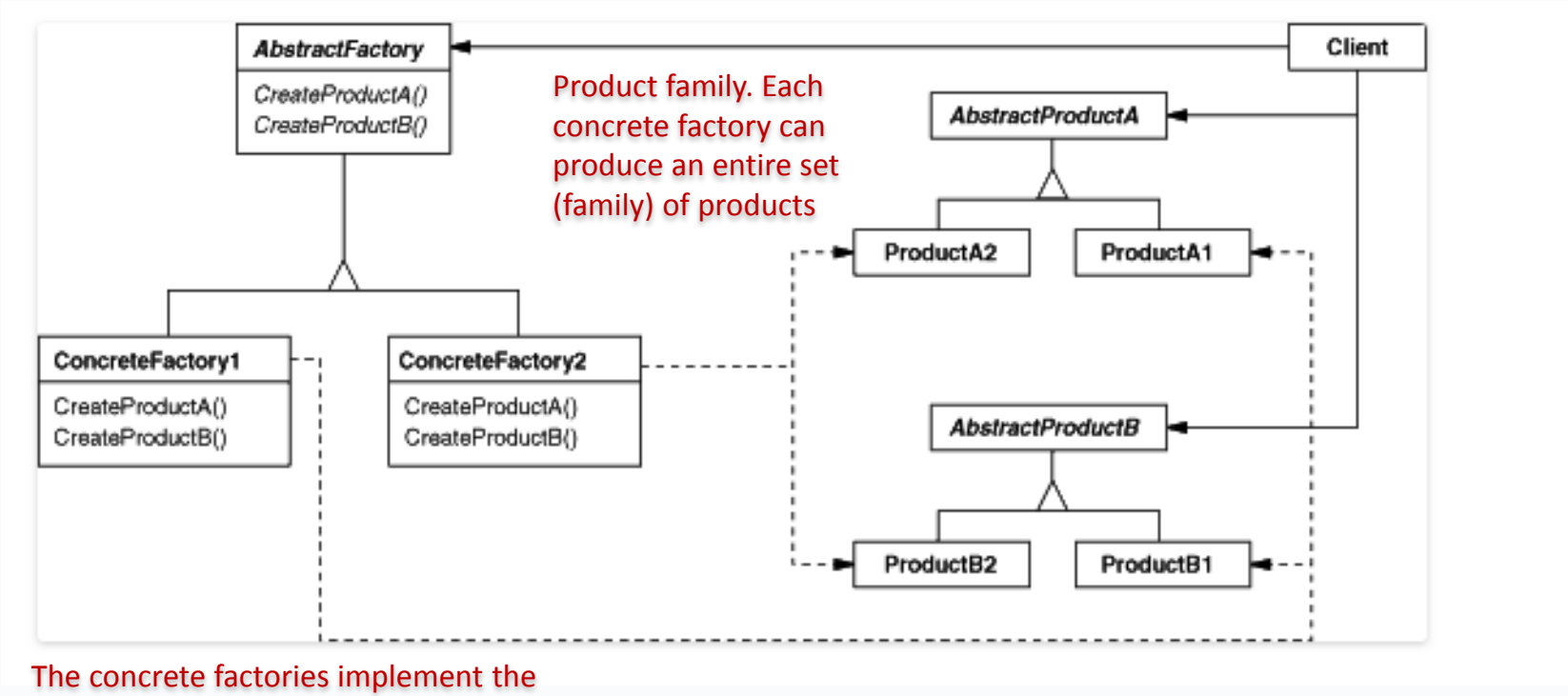
Intent

- A creational design pattern that that provides an interface for creating families of related or dependent objects without specifying their classes
- Very similar to the Factory Method pattern
  - The main difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.
- Actually, the delegated object frequently uses factory methods to perform the instantiation!

The Abstract Factory Method Pattern

Define an interface that all concrete factories must implement, which consists of a set (or family) of methods for producing the family of products

Client is written against the abstract factory and then composed at run-time with an actual factory



The concrete factories implement the different product families. To create a product family, the client uses one of these factories, so it never needs to instantiate the Products directly

The Builder Pattern

Motivation

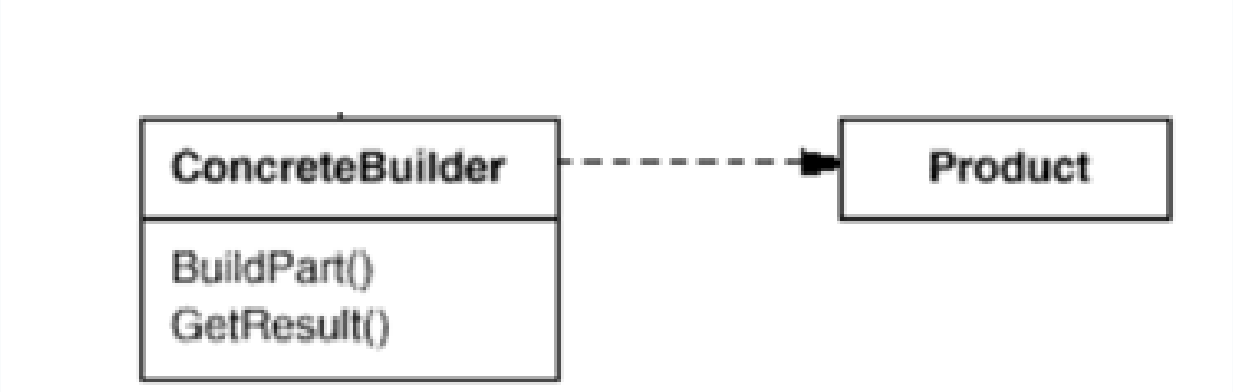
- A complex object that requires laborious step by step initialisation of many fields and nested objects (composite)
  - Avoid a “telescopic constructor” - one giant constructor with all possible parameters
  - overload a long constructor and create several shorter versions with fewer parameters (they will still call the main constructor, but pass in default values to omitted parameters)

The Builder Pattern

Intent

- The Builder Pattern enables encapsulation of the construction of a product and allows the object to be constructed in steps

  – Removes the needs for a “telescopic constructor” by building objects step by step. You use only the required steps (skip the optional ones)



Case Scenario

- Consider a class `User` with a large number of attributes where
  - you need to make the class immutable; all the attributes are declared final, and can only be set through the constructor
  - some of the attributes in the class are optional

```
public class User {
   private final String firstName;   //required
   private final String lastName;    //required
   private final int age;            //optional
   private final String phone;       //optional
   private final Address address;    //optional
}
```

Possible Options (1)

- One constructor that takes all possible attributes
- Overloaded constructors with reduced parameters, that still calls the main constructor, passing in default values to the optional parameters
- What are the issues with this “telescopic constructor” approach?

What are the issues?

- Code is harder to read and maintain, as number of parameters increase
- As a client, which constructor should I invoke?
- What is the default value, for the optional parameters?

```
...
public User(String firstName, String lastName) {
      this(firstName, lastName, 0);
}
public User(String firstName, String lastName, int age) {
      this(firstName, lastName, age, "");
}
public User(String firstName, String lastName, int age, String phone) {
      this(firstName, lastName, age, phone, "");
}
public User(String firstName, String lastName, int age, String phone, String
address) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
      this.phone = phone;
      this.address = address;
} ...
User u = new User(“john”, “doe”, 60, “888”, “”)
```

Possible Options (2)

- Use the Java Beans approach – i.e. use the default "no-arg" constructor and provide getters and setters for every attribute
- What's wrong with this approach?

- For an object instance that needs all the attributes, this relies on the client to invoke all the setter methods, otherwise object could be in a inconsistent state partway through its construction
- The JavaBeans approach also means that the class is no longer immutable

```
...
        public User() {}
        // setter and getter methods
        public void setFirstName(String name) {
           this.firstName = firstName;
        }
        public void setLastName(String name) {
           this.lastName = lastName;
        }
        // other methdos for age, phone, address} ...

        User u = new User();
        u.setFirstName("John");
        u.setLastName("Doe");
```

Builder Pattern

- Create a class UserBuilder that defines all the attributes as in the class User

- Create a constructor in UserBuilder for each attribute

- Create a method that returns the final created product - User

```
        ...
public static class UserBuilder {
   private final String firstName;
   private final String lastName;
   private int age;
   private String phone;
   private String address;

   public UserBuilder(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
   public UserBuilder age(int age) {
      this.age = age;
      return this;
   }
   // define additional constructors for each attribute phone, address…
   public User build() {
      return new User(this);
        }
   }
```

Integrating UserBuilder with class User

The class User is now immutable:

- All the attributes in the class User are final
- The constructor is defined as private in class User
- Only getter methods are defined and no setter methods
- Define UserBuilder as a nested static class inside the class User. Making the class static enables it to be instantiated without an instance of the outer class

```
public class User {
  private final String firstName;
  private final String lastName;
  private int age;
  private String phone;
  private String address;

  private User(UserBuilder builder) {
     this.firstName = builder.firstName;
     this.lastName = builder.lastName;
     this.age = builder.age;
     this.phone = builder.phone;
     this.address = builder.address;
  }
  public String getFirstName() {
         return firstName;

  // Define other getter methods...

  public static class UserBuilder {
     ...
     // Implement UserBuilder class as shown on previous slide
  }
```

Creating client code with the revised User class

```
User user1 = new User.UserBuilder("Jhon", "Doe")
      .age(30)
      .phone("1234567")
      .address("Fake address 1234")
      .build();
 ;
```

- The client code is more easy to read
- The state of an object is guaranteed to be in a consistent state
- A single builder can be used to create multiple objects by varying the builder attributes between calls to the “build” method

Use of Builder Pattern in Java API

- The Builder pattern is widely used in Java core libraries
  - `java.lang.StringBuilder#append()`
  - `java.lang.StringBuffer#append()`

```
// Use of Builder Pattern in StringBuilder
public class StringBuilderExample {
  private StringBuilder builder = new StringBuilder(64);

  public void append(String content) {
  if (content != null) {
          builder.append(content);
  }
}

  public static void main(String[] args) {
    StringBuilderExample e = new StringBuilderExample();
    String text = "It's a rainy day.";
    e.append(text);
    text = "I will need an umbrella";
    e.append(text);
    System.out.println(e.builder.toString());
  }
}
```

Builder Pattern and Law of Demeter

- The Builder pattern is based on an idiom “Fluent API” and makes use of chaining of methods
  - Does this imply that the builder pattern violates the Law of Demeter (The principle of least knowledge) ?

Benefits of a Builder Pattern

- The Builder pattern is flexible
  - A single builder can be used to build multiple objects. The parameters of the objects can be tweaked between object creations
  - A client can pass such a builder to a method to enable the method to create one or more objects for the client. For this, need a type to represent the builder

```
// A Builder for objects of type T
public interface Builder<T> {
   public T build();
}

// The UserBuilder could be declared to implement
// Builder<User>
```

The Builder Pattern as defined in Gof

Motivation

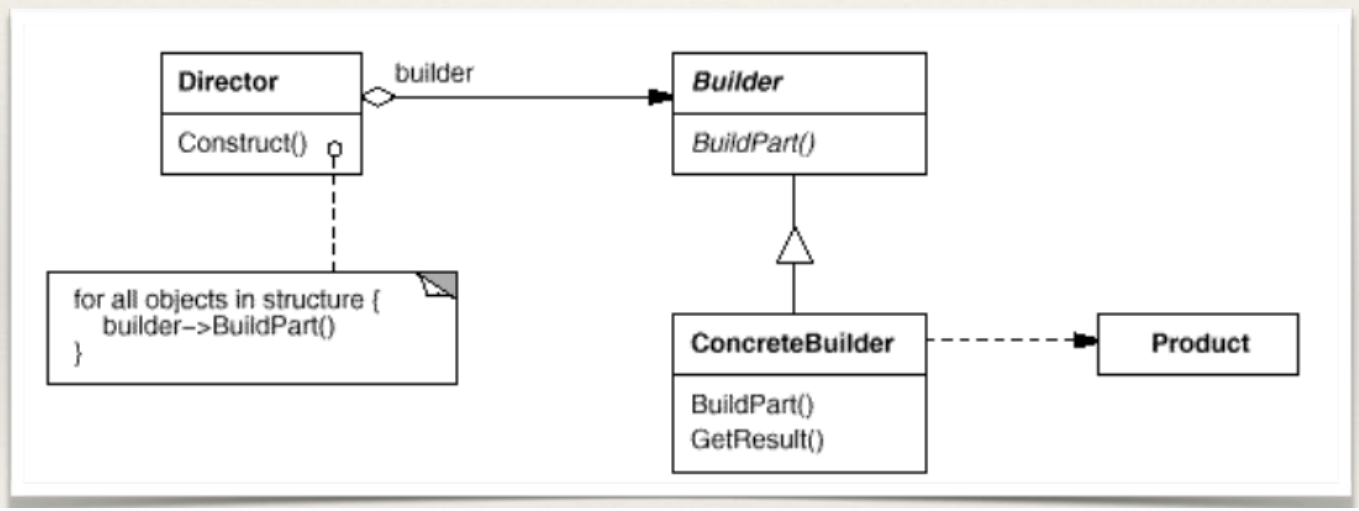
- Create different representations of a product (e.g., a house with a swimming pool, a house with a porch) or products that have similar build steps but differ in details
  - Avoid creating sub-classes for each different representation

Intent

- The Builder Pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.
  - Enables creation of different representations of a product using the same building process, where each distinct product will be represented by a separate builder class. Code that controls the construction order may live in a single directory class
  - Supports a much easier construction of a composite tree structure

Builder Pattern: UML class diagram

- Create **`Builder Interface`** and define the production steps (must include the common steps, and variations)
- Create a concrete **`Builder`** class for each product representation (and implement the steps)
- Think about creating a **`Director`** class. Its methods should create different product configurations, using different steps of the same builder instance.



Disadvantages of Builder Pattern

To create an object, you must first create its builder

- Cost of creating the builder could be a problem in some performance-critical situations
- The Builder pattern is more verbose than the telescopic constructor pattern, so it should only be used when there are say, four or more parameters

Summary of the Builder Pattern

- Use the Builder Pattern:
  - When the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - When designing classes whose constructors would have more than a handful of parameters, and especially if many of these were optional
  - When the construction process must allow different representations for the object that is constructed

Singleton Pattern

Singleton Pattern

Motivation

- Need to ensure that only a single instance of a class is created and made available to the entire application e.g., access to a shared resource, such as a database
- Ensure, additional instances cannot be created
- Provides a global access to this single instance

Intent

- A creational design pattern, that only a single instance of a class is created and provides access to this single instance by
  - Making the default constructor private.
  - Creating a static creation method that will act as a constructor. This method creates an object using the private constructor and saves it in static variable or field. All following calls to this method return the cached object.

Singleton Pattern (Naïve, single-threaded version)

```
public final class Singleton {
        private static Singleton instance;

        // private constructor
                private Singleton() {
        }

        public static Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
        } return instance;
}
```

- The above naïve implementation is not thread-safe i.e. if the above implementation is used by multiple threads, it cannot be guaranteed that only a single instance of the class is created
- Need to take additional measures to support multi-threaded applications (refer week 11 lecture on using synchronized to protect current access to shared resources)