# Design Patterns

Factories

Strategy Pattern

Builder Pattern

# Design Patterns

- First initiated in *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides (referred to as the Gang of Four or GOF).
- An approach that enables reuse of software at the design level

- We know it’s good to reuse code (by invoking methods, for example).
- Design patterns allow us to reuse on a bigger level- to reuse the *approach* to a solution.

# Design Patterns

- Design patterns provide a shared vocabulary.
- Design patterns represent best practices.

- Design patterns help us build programs that are:
  - Reusable
  - Extensible
  - Maintainable

- The GOF based their design patterns on the following two principles of object-oriented design:
  - Program to an interface, not an implementation.
  - Favor object composition over inheritance.

# Changing Our Ways

- Foundational object-oriented principals are still key!
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism
- These are building blocks. It’s critical to learn them first.
- But we now might start to do things a little differently…
  - We must learn to walk before we can learn to run!

# FACTORIES

# Practice

- Use the simplified Employee classes.
- Write a Department class that creates Employees based on user input and adds them to a list for that department.

# Is our solution flexible?

- What if we suddenly have interns in addition to full time and part time employees?
- We'd have to update the add method in the Department class…
  - But also **all other classes** where Employee objects are created. That could be a lot of places!

# Factory Classes and Methods

- A factory method creates objects.
  - Usually static, but not always.
  - When included, often change the constructor to private.

- A factory class contains a collection of factory methods.

- You might have used factory methods:
  - NumberFormat.getCurrencyInstance()
  - NumberFormat.getPercentInstance()
  - Calendar.getInstance()
  - myArrayList.iterator()

# Key Idea

- Identify what varies and separate it from what stays the same.
- Separate object creation from object processing.
    - *decoupling*

# Static Factory Methods- Design

- You have a parent/base class (usually abstract) (or could also be an interface).
- You have several child/sub classes (or several classes that implement the interface).
- The factory method returns the parent/base class.
  - Factory method decides which child class to instantiate.
  - Could do so using user input or a parameter.
- Common names
  - valueOf
  - of
  - getIntance
  - newInstance
  - getType
  - newType

# Practice

- Write a simple static factory method class to create Employee objects.

# Is our solution flexible?

- Much better! We’ve separated the creation of the object from the processing of the object.
  - Department has no knowledge of what kind of Employee it is processing. It doesn’t need to know!
  - In the future, we can update just the factory method if we want to add new subclasses.

# Benefits of Factories

- Can choose from multiple child classes and return a subtype
  - The client only needs to know about the parent class functionality- it doesn’t need to know the full class hierarchy
- Can reuse objects (e.g., database connections)
- Can return null
- Can have descriptive names and can support different interpretations of the same parameter types
  - MyClass.newInstanceByID(int id)
  - MyClass.newInstanceBySize(int size)
- Drawbacks
  - Classes with private constructors cannot be extended.
  - Factory methods are not always easily identifiable.

# On Your Own Practice

- Add one or more static factory methods to the Store Inventory classes you created in Modules 01 and 02.
  - Consider what “type” characteristics you want to use to create your items.
  - Consider whether you want to use a method or class.
- Update your driver/tester program to test out your factory method.

# The Factory Pattern

- Note that a simple static factory method or a factory class is **not** the “factory pattern” as defined by the GOF.
  - But is shares similar characteristics: it separates object creation from object processing.

# The Factory Pattern

- For our example: what if different departments want to create different kinds of employees?
  - Sales Department wants to create only part time.
  - IT Department wants to create full time, contract, and intern.
  - We would need different factories!
  - But then how would the department know which factory to use?

- The solution is to:
  - include an abstract factory method in the parent “grouping” class
  - include an implemented factory method in each “grouping” subclass

```
abstract Thing createThing(String type)
```

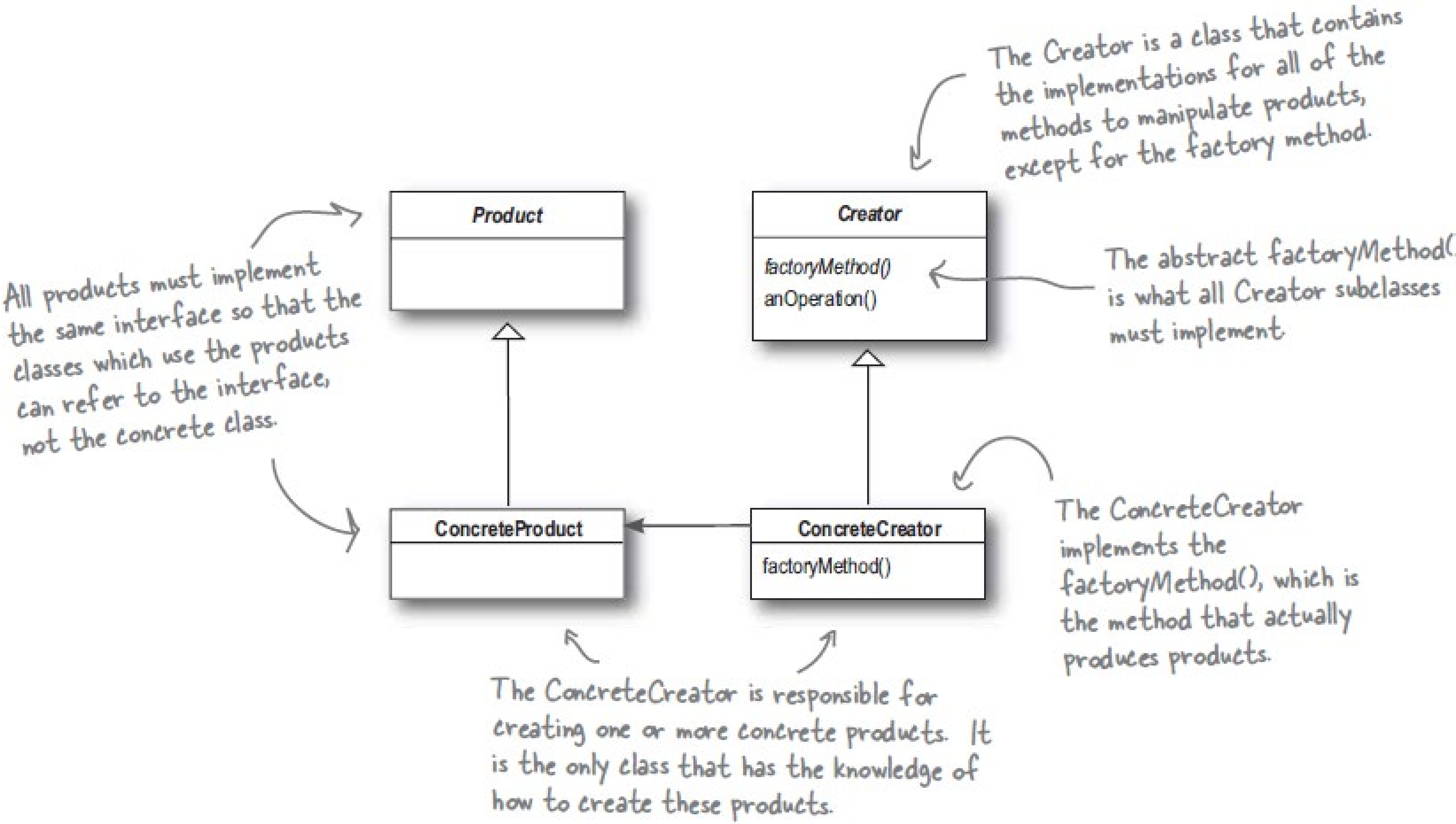
# The Factory Pattern

- The factory method handles object creation (often in a child class) and separates that behavior from the processing of objects (often in the parent class).
  - *Decoupling*

- The product classes (“Thing”) are produced by the factory method.
  - In our example: FullTimeEmployee, PartTimeEmployee

- The *creator* class never knows what kind of concrete product object is being used… and that’s good!
  - In our example: Department class doesn’t know what kind of employees are being processed- it doesn’t need to know!

# Factory Pattern




From Head First Design Patterns, page 134.

# Our Example

- Employee is the *Product* class
- FullTimeEmployee and PartTimeEmployee are the *Concrete Product* classes.
  - We could also think about adding Intern, TravelingSalesperson, ContractEmployee, etc.
- Department is the *Creator Class*.
  - This class is in charge of processing or manipulating the products (employees).
  - In our case, this means running payroll, benefits, and review, and then adding the employee to the list.
  - `public abstract Employee createEmployee();`
  - `public void processEmployeeList() { … }`
- SalesDepartment and ITDepartment are the *Concrete Creator* classes.
  - These classes are in charge of actually creating the products (employees).
  - Each department knows exactly what kind of employees it can create.
  - These classes implement the createEmployee method.
  - ```
    public Employee createEmployee() {
      // obtain type information and create the appropriate types
    }
    ```
    ```

# Is our solution flexible?

- Yes! Each department decides how to create its employees. But all employees are processed in the same way.
  - If we wanted to really guarantee this, we could make the processing methods final.

# Not every problem needs a factory pattern solution!

- When might I consider this pattern?
  - If your concrete product classes are likely to change- i.e., concrete products often added, removed, etc.
  - If different creator classes create different kinds or combinations of products.
- If this situation describes the problem, the factory pattern might be a good solution.

- Note: if you want to separate object creation from object processing, but all of your creators use the *same* combinations of objects (e.g., all departments have the same options for employees), then a simple factory method/class might be a better solution. The full factory pattern might not be necessary.

# On Your Own Practice

- Use the Store Inventory classes from Modules 01 and 02.
  - These classes describe the *product* and *concrete product*.
- Write an abstract class for a department or division within the store that will sell your products.
  - This is the *creator* class.
  - Use the factory pattern by writing an abstract factory method.
- Write two child classes that inherit from the abstract creator class.
  - These are the *concrete creator* classes.
  - Implement the factory method.

# The Builder Pattern

# Practice

- Review the Address example.

# Is our solution easy to use?

- Not really!
  - Telescoping Constructors
- What are all those Strings?? What is the order?
- Can’t get all the combinations I want!

# JavaBeans Pattern

- Invoke a default constructor and then call setters.

- Pro: clear to understand
- Con: object could be partially initialized
- Con: objects cannot be made immutable
  - An immutable object is an object whose state cannot be changed
  - We'll need these when we discuss concurrency!

# The Builder Pattern

- Create a *builder* class that contains setter-like methods to specify all the information. Then build the object through the builder.

```
public static class ThingBuilder
   public ThingBuilder (required params)
   public ThingBuilder optionalParam(type var)
   public Thing build()
```

# Practice

- Add a builder to the Address class.

# Is our solution easy to use?

- Yes!

```
User u = new User(“John”, “Doe”, 30, 145,
“555-5555”, 555, “Main St”);
```

vs

```
User u = new User.Builder(“John”, “Doe”)
     .age(30)
     .id(145)
     .phone(“555-555”)
     .address(555, “Main St”).
     .build();
```

# Using the Builder Pattern

- Add a `public static Builder` class inside your class.
- Add private variables that mirror the instance data variables.
  - Use default values when appropriate.
- Make required instance data variables parameters to the Builder constructor.
  - Or, check their values and throw an exception before returning the Thing.
- Provide methods for all other optional instance data variables.
  - These methods update the builder's data and return the builder object.
- Write a `build` method.
- Make the class constructor private and accept one parameter of type Builder.
  - Use the builder variables to initialize the instance data variables.

# Validity Checks

- The build method can check values and throw an `IllegalStateException` if a value is invalid or missing.

# Benefits of the Builder Pattern

- Simplifies telescoping constructors
- Allows creation of immutable objects
- Allows multiple var-args (one for each method)

- The Builder pattern is a good choice when you have a constructor with multiple parameters, especially with repeated types and when many are optional.

# On Your Own Practice

- Add a builder to one of your Store Inventory classes.

  OR

- Add more variables to one of the Employee classes and add a builder to that class.
  - Bonus: add an Address object as instance data variable- builder using a builder!

- Include at least one validity check in the build() method.
- Write code in a tester program to use the builder.

# The Strategy Pattern

# Practice

- Implement the pay() method in the Employee classes.
- Full time employees are paid via salary.
- Part time employees are paid via hourly.
- Interns are not paid.

- What if we had other employee types that were also paid via salary?
- What if part-timers could be paid either by salary or by hourly?

# Is our solution maintainable?

- Problem: Multiple child classes have the same implementation.
  - Repeated code is bad!
- Problem: Child classes might have complicated conditionals to determine action.

- Solution? Put the implementation in the parent class.
  - No good: Not *all* child classes have this implementation. So this is logically not accurate.

- Solution? Add another layer of classes and put the implementation there.
  - No good: Could lead to many, many classes all to support one method.

- Solution? Separate out the functionality into an interface.
  - No good: We still have the duplicate code in the classes that implement the interface!

# The Strategy Pattern

- Create an interface to represent the functionality.
- Add an *instance of* the interface into the parent or child classes.
- Implement the interface in various concrete classes.

# Key Ideas

- Separates what varies from what stays the same.
- Capture commonalities to avoid duplicate code.
- Program to an interface, not an implementation.
- Favor composition over inheritance.

# Practice

- Create a Payer interface.
- Write classes that implement this interface.
  - These classes represent the various ways to pay.
- Add an instance Payer to the Employee class.
- Implement a runPayroll method in Employee.
- Create the Payer objects in the child classes.

# Is our solution maintainable?

- Yes!
- We’ve written each method of payment only once.
- We can easily write new methods of payment if they come up.
- Each child class initializes the kind of payer that it needs.

- We could even set the functionality at runtime!

```
public void setPayer(Payer p)
```

# Key Ideas

- Separates what varies from what stays the same.
  - All employees are paid *somehow* (that's the same- leave that functionality in Employee by having an instance data variable)
  - *How* they get paid varies (that's different- separate it out into the Payer interface and the classes that implement Payer and by instantiating the variable in the child classes)
- Capture commonalities to avoid duplicate code.
  - SalaryPayer is written only once. We can then use it as many times as we want- just create a new SalaryPayer object!
- Program to an interface, not an implementation.
  - The method that runs payroll doesn't contain the implementation of how each employee is paid. It just uses a reference to an interface object which will hold the proper information on how to pay.
- Favor composition over inheritance.
  - Inheritance means leaving the pay method implemented/ overridden in Employee and child classes.
  - Composition means having an *instance of* a Payer object in the Employee class.

# When to Use Strategy Pattern

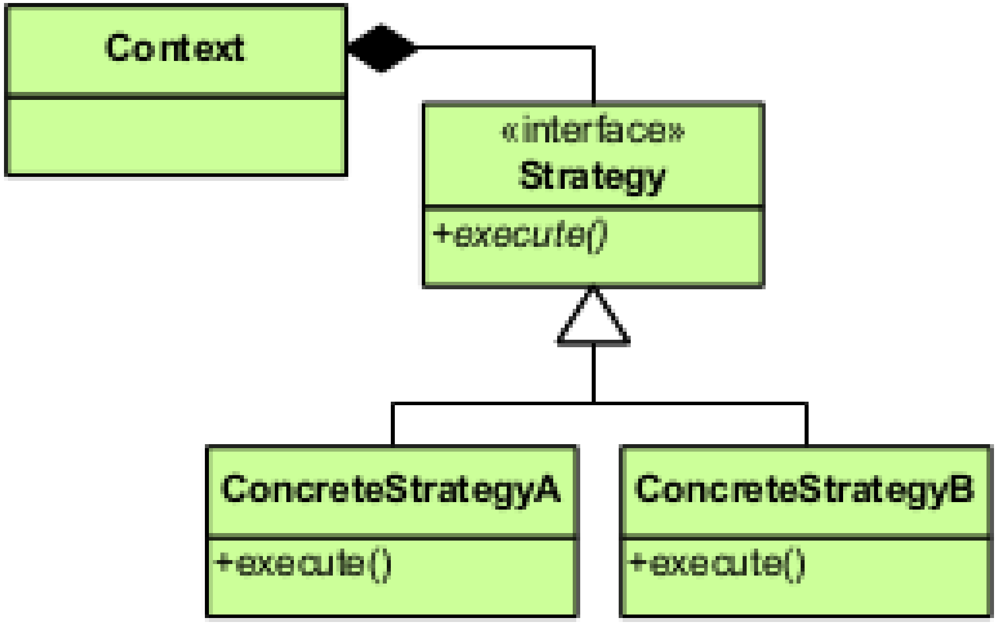
- When you have several different behaviors that your child classes will choose from.
  - Some classes will use the same behavior (which would normally result in repeated code).
  - The behavior might need to be chosen dynamically (objects of the same type might need different behavior).

- Use this pattern to reduce duplicated code and reduce long lists of conditionals that try to figure out behavior.

# Using the Strategy Pattern

- Create an interface that contains the behavior.
- Write classes that implement the interface.
  - These classes represent the various *strategies* for implementing that behavior.
- Add an instance of the interface into classes that have this behavior.
  - Instantiate the instance with the appropriate class.
  - Write a method that invokes the method from the interface.

# The Strategy Pattern




*From Wikipedia*

# On Your Own Practice

- Design a strategy pattern into your Store Inventory classes.

# COMPARATOR

# The Comparable Interface

- Provides a way to order objects.
  - The *natural ordering*
- Used by Arrays.sort and Collections.sort.

```
public MyClass implements
        Comparable<MyClass> {…

public int compareTo(MyClass obj)
```

# The Problem

- Sometimes I want to sort based on ID. Sometimes I want to sort based on Name.
- I can only have one compareTo method!

- Strategy pattern to the rescue!
  - Different behaviors that objects can take- possibly chosen at runtime.

# The Comparator Interface

- Allows you to specify an ordering of two objects.

```
public int compare(T o1, T o2)
```

- API: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

# Writing a Comparator Class

- Comparators often written as static inner classes.
- Common also to create constants of type Comparator<T> that are instances of the comparator.

```
public final static
   Comparator<Employee> ID_ORDER
    = new EmployeeIDComparator();

public final static
   Comparator<Employee> NAME_ORDER
   = new EmployeeNameComparator();
```

# Writing a Comparator Class

```
private static class MyClassComparator implements Comparator<MyClass> {
        public int compare(MyClass m1, MyClass m2) {
                // compare m1 and m2 using their instance data;
                // common to invoke compareTo on instance data variable

                // return negative number if m1 < m2, positive if
                // m1 > m2, and 0 otherwise
        }
}

public static final MyClassOrder MY_SORT = new MyClassComparator();
```

# Using a Comparator Object

- Collections.sort(myCollection, myComparator)
- Arrays.sort(myArray, myComparator)

# Comparator as Strategy

- There are many different ways to order objects for sorting
- Separate ordering functionality into an interface (Comparator)
- Implement that interface with multiple concrete classes (in this case, often nested classes) that implement *how* to compare
- Choose how we want to compare (and sort) by different methods at different points in time (or at runtime!)

# Practice

- Add a comparator to the Employee class.
- Create a list of employees and test out different orderings.

# Comparable vs. Comparator

| | **Comparable** | **Comparator** |
|---|---|---|
| **What is represents** | The *natural ordering* of an object<br><br>Only one per class | Any ordering<br><br>Can have as many as you want |
| **Where it is written** | Implemented by the object's class:<br><br>`public MyClass implements Comparable<MyClass>` | Implemented by a new (often nested) class:<br><br>`public MyComparatorClass implements Comparator<MyClass>` |
| **The method** | `public int compareTo (MyClass otherObj)` | `public int compare (MyClass object1, MyClass object2)` |
| **How method is invoked** | `myObject.compareTo( otherObject)` | `myComparator.compare( thisObject, thatObject)` |
| **Used in sorting** | By default:<br><br>`Collections.sort(myCollection);`<br><br>`Arrays.sort(myArray);` | By specifying:<br><br>`Collections.sort(myCollection, new MyComparatorClass());`<br><br>`Arrays.sort(myArray, new MyComparatorClass());` |

# Practice

- Review the user examples.
- Implement the various comparators.

# On Your Own Practice

- Create two comparator classes for your Store Inventory. Practice sorting by different characteristics.