your design problems

- inter-dependencies?
  - coupling between what you have completed and what new features you wish to add?
  - having to rewrite working code just because of some far-removed change?
  - extensibility? (inordinate effort needed to just add the next variation on some theme)
  - migrating responsibility from one class to another? (realizing methods belong in different classes)
  - changing the signature of one method and having to track down all consequences across classes?
- identify the aspects of your application that vary and separate from what stays the same
- strive for loosely couple designs between objects that interact
- what else? By now you have some experiences to tell
extensibility with the different design patterns

- The **Strategy** Pattern
  - adding new strategies for a given hierarchy
  - adding new subclasses to hierarchy
- The **Observer** Pattern
  - adding new Observables (separate hierarchies of Observables versus siblings versus subclasses)
  - adding new Observers
- The **Decorator** Pattern
  - adding new Decorators
  - dealing with identity issues (am I a suit or what?)
- The **State** Pattern
  - adding new States
  - adding new State-dependent methods
  - adding new classes that share States
- The **Visitor** Pattern
  - adding new visitors
  - adding new visitees
- in general, coupling between delegate and parent
what words of wisdom thus far?

• identify the aspects of your application that vary and separate from what stays the same

• program to an interface, not an implementation

• favor composition over inheritance

• strive for loosely couple designs between objects that interact

• minimize your daily consumption of simple carbohydrates
the factory pattern

• when you see “new”, think “concrete”
  
  BaseType t = new SpecificType();

• if different types of concrete objects, one gets:
  • case-specific code (conditionals, etc.)
  • future maintenance nightmares

• breaks caveat “do not program to an implementation”

• not “closed to modification” as additional classes need to be added

• instead, program to an interface
  • make it open to extension
  • isolate the aspects that vary from those that stay the same

• the Factory Pattern is commonly used in applications where there is the API exposes too much concrete detail
creating a new instance of some product

- suppose you have a class hierarchy and need a method to create new instances of the concrete classes:
  1. perform decision process to determine which specific subclass (product) to instantiate
  2. initialize the specific instance, with parameters to constructor
  3. perform some common initial operations on the instance
  4. return the instance (deliver the product)

- the problems arise when new concrete classes are added to the hierarchy
- problems are compounded when each subclass has its own idiosyncrasies in creating new instances of some product
• program to an interface, not an implementation

• “if your code is written to an interface, then it will work with new classes implementing that interface through polymorphism”

• “if your code makes use of lots of concrete classes, ... that code will have to be changed as new concrete classes are added” (i.e., not “closed for modification”)

public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza p = null;

        if (style.equals("NY") {
            if (type.equals("cheese")) {
                p = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                p = new NYStyleVeggiePizza();
            }
        } else if (style.equals("Chicago") {
            if (type.equals("cheese")) {
                p = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                p = new ChicagoStyleVeggiePizza();
            }
        }
        p.prepare()... bake()... cut() ... box()...
        return p;
    }
}

incidentally, what pattern would help clean up this mess?

```java
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza p = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                p = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                p = new NYStyleVeggiePizza();
            }
        }
        if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                p = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                p = new ChicagoStyleVeggiePizza();
            }
        }
        p.prepare()... bake()... cut() ... box()...
        return p;
    }
}
```

• yep, the **Decorator Pattern**.
• but even with using the Decorator, **there is still dependency between the store and the objects it creates and delivers**
so what’s wrong with that?

- the pizza store is very dependent upon all the objects it creates
  - it creates them directly within the store class
  - add one new style or change the interface to accept new types for a given style ... trouble.
- “Because any changes to the concrete implementations of pizza affects the PizzaStore, we say that the PizzaStore ‘depends on’ the pizza implementations.” (p.138)
- reducing dependencies to concrete classes “is a good thing” (“Dependency inversion Principle”)

Depend upon abstractions. Do not depend upon concrete classes ... to be elaborated upon later.
public Product order(String type) {
    Product p;

    if (type.equals("type1"))
        p = new Type1Product();
    else if (type.equals("type2"))
        p = new Type2Product();
    ...
    p.prepareForDelivery();

    return p;
}

• isolating what changes from what stays the same

not "closed for modification"
public Product order(String type) {
    Product p;

    p.prepareForDelivery();

    return newInstance;
}

- separate the two aspects into different methods
  - creating an instance of some Product
  - order fulfillment
now, make a “simple factory” (p.115)

public class Factory {

    public Product create(String type) {
        Product p;

        [creates a specific type of Product]

        return newInstance;
    }

}

• many different clients might use Factory
• the Factory can create different types of Product
• the factory might be a static class (not instantiated)
• this “simple factory” is not yet the Factory Pattern
next, use the “simple factory” (p.116)

- a Store would be one client that uses the factory:

```java
public class Store {
    private Factory factory;

    public Store(Factory f) {
        factory = f;
    }

    public Product order(String type) {
        private Product p = factory.create(type);
        prepareForDelivery(p);
        return p;
    }
}
```
usage

• create a factory to pass to the store
  Factory f = new Factory();
• create a store that uses this factory
  Store s1 = new Store(f);
• then order and deliver Products from the store
  Product p1 = s1.order("standard");
  Product p2 = s1.order("special");
• one might have have multiple subclasses of factory
  CheapFactory c = new CheapFactory();
  Store s2 = new Store(c);
  Product p3 = s2.order("special");
• or the store could change factories on its own (like, “we had to change suppliers to keep costs down”)
• nothing keeping one from going directly to the factory and calling create directly on the factory
now, preparing for the real Factory Pattern

• recall that create was taken out of the store class and put into a factory (outsourced, so to speak)
• instead, put create back into the store, but as an abstract method

```java
public abstract class Store {

    public Product order(String type) {
        Product p = create(type);
        p.prepareForDelivery();
        return p;
    }

    abstract protected Product create(String type);
}
```

• now it’s up to the concrete subclasses of Store to dictate the actual product creation
public abstract class Store {

    public Product order(String type) {
        Product p = create(type);
        p.prepareForDelivery();
        return p;
    }

    abstract protected Product create(String type);
}

• the method order is public in the base class Store.
  • when the store receives an order, it privately (or "protectedly") calls the abstract method create
  • create is now a Factory Method
  • i.e., the store delegates the specifics of creating to its concrete subclasses
public abstract class Store {

    protected Product order(String type) {
        Product p = create(type);
        p.prepareForDelivery();
        return p;
    }

    abstract protected Product create(String type);
}

public class UpscaleStore extends Store {
    protected Product create(String type) {
        Product p = [makes an expensive Product]
        return p;
    }
}

public class CheapStore extends Store {
    protected Product create(String type) {
        Product p = [makes a cheap Product]
        return p;
    }
}
the Factory Method (p.121-125)

• the public can only order through the store
• you choose the store; the store chooses the factory
  • depending on the specific store, a factory method then 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.
• a factory method might be parameterized (e.g., the String type) to provide variations on what is created.
• driver code wants a particular style of pizza:

```java
PizzaStore nyPizzaStore = new NYPizzaStore();
nyPizzaStore.orderPizza("cheese");
```

• the act of ordering the pizza results in calling the subclass NYPizzaStore’s version of createPizza

```java
Pizza pizza = createPizza("cheese");
```

• while you went into a generic, abstract PizzaStore, and ordered a cheese pizza. It happened to be a NYPizzaStore and that’s why you got a NY style pizza.
the Factory Method Pattern

- The various stores are **Creator Classes**
  - base class is the “abstract creator class”
  - subclasses are “concrete creators”
- The products have an independent hierarchy
  - variations in type within product line from a given concrete creator (vanilla, strawberry, ...)
  - variations in concrete creators (Prince Pücklers, ...)
- “The Factory Method Pattern” defines an interface for creating an object, but lets the subclasses decide which class to instantiate.
discussion of the Factory Method

• useful even if there is only one concrete creator class
  • decoupling implementation of the product from its use (the clientele that visit the store)
  • can add concrete creators
  • can change product ("creator not tightly coupled to any concrete product") ... really?
• the factory method is not publicly available:
  • “simple factory” had public create method (retail)
  • the factory method is protected, under control of the store only (wholesale only, not retail)
  • the factory method could be concrete in the basic store class, and overridden (shadowed) by some subclasses of store.
• regarding types, they suggest other than strings, to use enums.
inversion principle

• no variable should “hold a reference” to a concrete class
  • if you use new, you are holding a reference to a concrete class; use a factory to get around that.
• no class should derive from a concrete class
  • if you derive from a concrete class, you are depending on a concrete class. Derive from an abstraction (abstract class or interface) instead.
• no method should override an implemented method of any of its base classes
  • if you override an implemented method, then the base class was not really an abstraction
  • any implemented methods should be shared by all subclasses.
• at least, try to minimize violations of the above
the next step... a factory for the ingredients

public interface IngredientFactory {
    public IngredientA getIngredientA();
    public IngredientB getIngredientB();
    ...
}

• Make concrete “ingredient factories” which supply the ingredients for various concrete factories.

public class ExpensiveIngredientFactory implements IngredientFactory {
    public IngredientA getIngredientA(){ ...}
    ...
}

public class CheapIngredientFactory {
    implements IngredientFactory {
    public IngredientA getIngredientA(){ ...}
    ...
}
and using the IngredientFactories

abstract public class Product {
    abstract void prepare();
}

public ExpensiveProduct extends Product {
    private ExpensiveIngredientFactory f;

    public ExpensiveProduct(IngredientFactory f) {
        this.f = f;
    }

    public void prepare() {
        ... uses expensive ingredients
    }
}

• and likewise for CheapProduct
• the Product is now also abstract, hence the term Abstract Factory Pattern
  • this permits a recursive use of the factory pattern for creating the products that are sold by the stores
so finally

• the Factory Method Pattern
  ```java
  public class UpscaleStore extends Store {
      protected Product create(String type) {
          Product p = [makes an expensive Product]
          return p;
      }
  }
  ```

• versus the Abstract Factory Pattern
  ```java
  public class UpscaleStore extends Store {
      protected Product create() {
          private IngredientsFactory f = new ExpensiveIngredientsFactory();
          Product p = [makes a product using expensive ingredients]
          return p;
      }
  }
  ```