9. Creational Pattern

Creational Patterns

- Abstracts instantiation process
- Makes system independent of how its objects are
  - created
  - composed
  - represented
- Encapsulates knowledge about which concrete classes the system uses
- Hides how instances of these classes are created and put together
Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Example that would benefit from Abstract Factory

```
ComputerModelA
  MemoryType A CPUTypeA ModemTypeA
BuildComputer(ComputerModelA& comp)
{  comp.Add(new MemoryTypeA);
   comp.Add(new CPUTypeA);
   comp.Add(new ModemTypeA);  }
```

What if I want to build a Computer of Model B with Model B Memory, CPU and Modem?
Using Abstract Factory...

```c++
BuildComputer(Computer& comp, ComputerFactory& compFactory)
{
    comp.Add(compFactory.createMemory());
    comp.Add(compFactory.createCPU());
    comp.Add(compFactory.createModem());
}
```
When to use Abstract Factory?

- Use Abstract Factory when:
  - system should be independent of how its products are created, composed and represented
  - system should be configured with one of multiple families of products
  - a family of related product objects must be used together and this constraint need to be enforced
  - you want to reveal only the interface, and not the implementation, of a class library of products

Structure
Consequences of using Abstract Factory

- Isolates concrete classes
- Makes exchanging products families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also known as Virtual Constructor
Example that would benefit from Factory Method

We want to develop a framework of a Computer that has memory, CPU and Modem. The actual memory, CPU, and Modem that is used depends on the actual computer model being used. We want to provide a configure function that will configure any computer with appropriate parts. This function must be written such that it does not depend on the specifics of a computer model or the components.

Example using Factory Method

Memory

Computer

createMemory()
createCPU()
Configure()

Memory* mem = createMemory();
add (mem);
CPU* cpu = createCPU();
add(cpu);

return new MemoryA;

ComputerA

MemoryA

createMemory()
createCPU()

return new CPUA;
When to use Factory Method?

- A class can’t anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Structure

```
Product

ConcreteProduct

Creator

ConcreteCreator

FactoryMethod

FactoryMethod() => Product = FactoryMethod();

anoperation() => return newConcreteProduct;
```
Consequences of using Factory Method

- Provides hooks for subclasses
- Connects parallel class hierarchies

Factory Method Vs. Other Pattern

- Abstract Factory often implemented with Factory Method
- Factory Methods usually called within Template Methods
- Prototypes don’t require subclassing the Creator. However, they often require initialize operation on the Product class. Factory Method doesn’t require such an operation
- Proliferation of subclasses
Builder Pattern

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

---

Example that would benefit from Builder Pattern

A computer has memory and CPU. Building a Computer has two steps - building memory & building the CPU. However, one model has one memory unit & one CPU. Another has four memory units & two CPUs. How to write the code such that the same code can be used to build the two models of computers, & does not change if new models of computers are introduced.
Example using Builder Pattern

class ComputerBuilder{
    makeComputer();
    buildMemory();
    buildCPU();
};

class SimpleComputerBuilder :
    public ComputerBuilder {
    makeComputer();
    buildMemory();
    buildCPU(){comp->addCPU(new PROC);};
};

Example using Builder Pattern...
class SuperComputerBuilder :
    public ComputerBuilder {...
    getNumberOfProcessors();
    makeComputer();
    buildMemory();
    buildCPU(){
        comp->addCPU(new PROC);
        comp->addCPU(new PROC);
    }
}
Example using Builder Pattern...

Computer* CreateComputer(
    ComputerBuilder&
    compBlwr)
{
    compBlwr.makeComputer();
    compBlwr.buildMemory();
    compBlwr.buildCPU();
    return blwr.getComputer();
}

Example using Builder Pattern...

SimpleComputerBuilder
blwr;
Computer* comp =
    CreateComputer(blwr);

... SuperComputerBuilder blwr;
Computer* comp =
    CreateComputer(blwr);
...
blwr.getNumberofProcessors();
When to use Builder Pattern?

- Algorithm for creating complex object should be independent of the parts that make up the object and how they are assembled
- Construction process must allow different representation for the object that is constructed

Structure

```
Director
  ^
  \Construct()

builder

For all objects in structure {
  builder->buildPart();
}

Builder
  \buildPart()

Concreate Builder
  \buildPart()
  \getResult()

Product
```
Consequences of using Builder

- Lets product’s internal representation to vary
- Isolates code for construction & representation
- Gives finer control over construction process

Builder Vs. Other Patterns

- Builder takes care of complete creation of a complex product step by step
- Abstract factory emphasizes creation of families of products - one component at a time
- Builder builds a Composite
Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Example that would benefit from Prototype Pattern

There are several types of Memory available. A computer may have Memory Model I or Memory Model II (or any other model that may be created later). If I want a Computer that is equivalent to a given Computer, how do I create it?
Example that would benefit from Prototype Pattern...

Computer (const Computer & otherComp)
{
    memory = new MemoryI;
    *memory = *(otherComp.Memory);
    // Copy characteristics of the other
    // Computer’s Memory
}
What if other Computer had Memory Model II?

Example using Prototype Pattern

Computer (const Computer & otherComp)
{
    memory = otherComp->memory.clone();
}
When to use Prototype Pattern?

- System should be independent of how its products are created, composed and represented and
- the classes to instantiated are specified at runtime
- you want to avoid creating class hierarchy of factories that parallel the class hierarchy of products

Structure

```
Client
  Operation()
      prototype

Prototype
  clone()
      Concreate
      Prototype1
          clone()
      Return copy of self
```

p = prototype->clone()
Another Example of Prototype

- An application that stores objects to disk
- Making the load method extensible for new types of classes being added to the system

Consequences of using Prototype

- Adding & removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reduced subclassing
- Configuring application with classes dynamically
- Each subclass must implement \textit{clone} operation
Prototype Vs. Other Patterns

- Abstract Factory is a competing Pattern, however, may work together as well
- Composite & Decorator benefit from Prototype

Singleton Pattern

Ensure a class only has one instance, and provide a global point of access to it
Example that would benefit from Singleton Pattern

An application uses a Database Manager. It needs only one Database Manager object. It must not allow the creation of more than one object of this class.

class DBMgr {
    static DBMgr* pMgr;
private:
    DBMgr() { } // No way to create outside of this Class
public:
    static DBMgr* getDBMgr() // Only way to create.
    {
        if (pMgr == 0) pMgr = new DBMgr;
        return pMgr;
    }
};
DBMgr* DBMgr::pMgr = 0;

Usage:
DBMgr* dbmgrPtr = DBMgr::getDBMgr();
//Created first time called
When to use Singleton Pattern

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Structure

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Instance()</td>
</tr>
<tr>
<td>SingletonOperation()</td>
</tr>
<tr>
<td>GetSingletonData()</td>
</tr>
<tr>
<td>static uniqueInstance singletonData</td>
</tr>
</tbody>
</table>

return uniqueInstance;
Consequences of using Singleton

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations & representation
- Permits a variable number of instances
- More flexible than class operations

Singleton Vs. Other Patterns

- Several patterns are implemented using Singleton Pattern. For instance AbstractFactory needs a singleton pattern for single instance of the factory