# Design Patterns

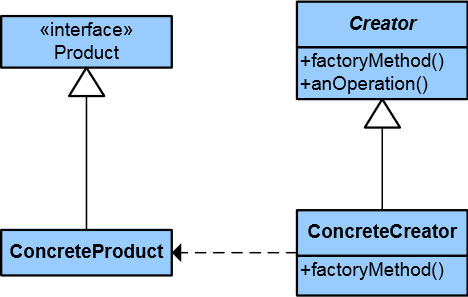
## Factory Method

**Motivation:** When working with a complex structure of classes, such as a system with many levels of inheritance, nesting, and dependencies, it is difficult to create a new instance of a class while keeping its state. A framework of documents, only has to care about what kinds of documents are in it, not what kind of document it should create. The **Factory Method** design pattern encapsulates the creation of documents and moves it out of the document structure.

**Intent:** Define an interface for creating an object but let the classes decide which class to instantiate while creating this object.

**Usages:** The **Factory Method** design pattern is used when a class does not know what object is should create and lets its inheritors specify this. Another common usage is when a concrete helper class is called depending on the type of the object.

**Structure:**

****

Each product implements a common public interface (or inherits from an abstract class) and has its own concrete creator makes an instance (overrides the factory method). The general creator, serving as a base class for all other creators uses the public interface and when called, turns to the concrete creators to find the one most suitable for the situation.

**Consequences:** The **Factory Method** design pattern makes instantiation much easier, also there is no need to know how a complex tree structure of classes works in order to create an object from it. Another thing is, the classes may call a helper class structure which has the same hierarchy as the main structure. The design pattern can be used to call the concrete helper class depending on the type of the object.

**Real-world examples:** There are many known uses of this pattern. It is particularly useful when there are complex hierarchies of classes and there should be a single interface to use them.

**C# example:**

// Product

abstract class Page { ... }

// Concrete products

class SkillsPage : Page { ... }

class EducationPage : Page { ... }

// Creator

abstract class Document

{

  private List<Page> pages = new List<Page>();

public Document()

  {

    this.CreatePages();

  }

  public List<Page> Pages

  {

    get { return this.pages; }

  }

  // Factory Method

  public abstract void CreatePages();

}

// Concrete creator

class Resume : Document

{

  // Factory Method implementation

  public override void CreatePages()

  {

    this.Pages.Add(new SkillsPage());

    this.Pages.Add(new EducationPage());

    this.Pages.Add(new ExperiencePage());

  }

}

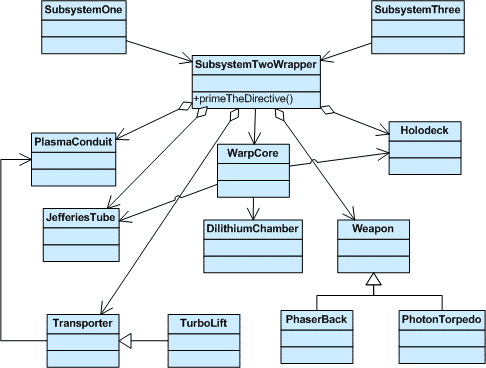
## Facade

**Motivation:** It is very convenient to separate a complex system into smaller subsystems. A general way to achieve this is to extract several commonly used operations in a separate class. The **Facade** design pattern introduces an interface which allows easy interaction with the subsystem.

**Intent:** Provide a unified interface to a set of interfaces in a subsystem. The **Facade** design pattern makes a complex system easier to use.

**Usages:** A common use for this design pattern is when a user needs a basic setup for a subsystem. When a system evolves, it becomes more customizable but also harder to use. The **Facade** design pattern provides users with an essential setup, leaving them to specify more options if needed. Another common use is to decouple a complex structure of classes with many inner dependencies from the other parts of a system.

**Structure:**

****

There is a complex hierarchy of classes and dependencies between them, but the facade (the class **SubsystemTwoWrapper** in this case) hides their complexity providing the option to perform a commonly executed task.

**Consequences:** The **Facade** design pattern has two main benefits. First, it makes a complex system easier to use, wrapping the complexity, and second, it provides internal decoupling between two subsystems, thus reducing the number of dependencies between them. The subsystems may communicate using their public interfaces.

**Real-world examples:** The design pattern is widely used. One common use is a compiler. A user does not need how the compiler performs its work, how it checks and translates the programming code to CPU instructions, etc. The compiler only gets a text file and produces an executable file.

**C# example:**

class Customer { ... }

// Subsystem class  
class Bank  
{  
 public bool HasSufficientSavings(Customer c, int amount) { ... }  
}

// Subsystem class  
class Credit

{

  public bool HasGoodCredit(Customer c) { ... }

}

// Subsystem class  
class Loan

{

  public bool HasNoBadLoans(Customer c) { ... }

}

// Facade

class Mortgage

{

  private Bank \_bank = new Bank();

  private Loan \_loan = new Loan();

  private Credit \_credit = new Credit();

  public bool IsEligible(Customer cust, int amount)

  {

    if (!\_bank.HasSufficientSavings(cust, amount))

    {

      eligible = false;

    }

    else if (!\_loan.HasNoBadLoans(cust))

    {

      eligible = false;

    }

    else if (!\_credit.HasGoodCredit(cust))

    {

      eligible = false;

    }

    return eligible;

  }

}

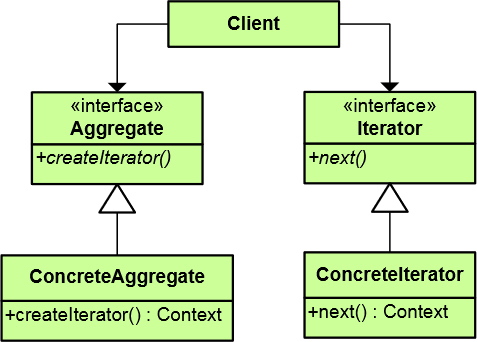
## Iterator

**Motivation:** Data structures, such as lists, queues, graphs, etc. need a way to expose their contents while keeping encapsulation. Also, they need to have some way of traversing. The **Iterator** design pattern takes the responsibility of the contents and the ways of traversal of a data structure. It also keeps track of the current element and what elements have already been traversed.

**Intent:** Provide a way to access the elements of an aggregate object (collection) sequentially without exposing its underlying structure.

**Usages:** The **Iterator** design pattern is used to access the contents of a data structure and to provide different ways to traverse it. It also supports polymorphic traversal, that is, an uniform interface for traversing different structures.

**Structure:**

****

The iterator defines a common interface for traversing the collection. The concrete iterator keeps track of the current position in the concrete case. The aggregate defines the public interface of data structures, while the concrete aggregate defines the concrete data structure. Used over the aggregate, the iterator provides a way to access elements of the structure.

**Consequences:** The **Iterator** design pattern provides easy changing of the traversal algorithm without modifying the original collection. To change the algorithm, only the iterator needs to be changed. It also extracts the logic for iterating over the collection so it does not need to implement its own logic for traversing.

**Real-world examples:** The design pattern is widely used in many ways when there are collections needing traversal. A really common example is the IEnumerable interface in C#.

**C# example:**

// Aggregate abstract class

abstract class Aggregate  
{

public abstract Iterator CreateIterator();

}

// Concrete aggregate

class ConcreteAggregate : Aggregate

{

  public override Iterator CreateIterator()

  {

  return new ConcreteIterator(this);

  }

public int Count { ... }

  // Indexer  
 public object this[int index] { ... }  
}

// Iterator  
abstract class Iterator

{

public abstract object First();

  public abstract object Next();

  public abstract bool IsDone();

  public abstract object CurrentItem();

}

// Concrete iterator  
class ConcreteIterator : Iterator

{

private ConcreteAggregate aggregate;

  public ConcreteIterator(ConcreteAggregate aggregate) { ... }

  public override object First() { ... }

  public override object Next() { ... }

  public override object CurrentItem() { ... }

  public override bool IsDone() { ... }

}