Spelling Checking & Hyphenation:

Goals:

- analyze text for spelling errors.
- introduce potential

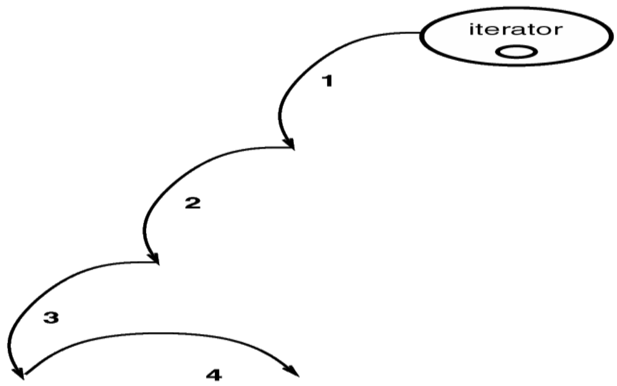
hyphenation sites. Constraints/forces:

- support multiple algorithms.
- don’t tightly couple algorithms with document structure.

Solution: Encapsulate

Traversal: Iterator

- encapsulates a traversal algorithm without exposing representation details to callers.
- uses Glyph’s child enumeration operation.
- This is an example of a “preorder iterator”.



TERATOR **object behavioral**

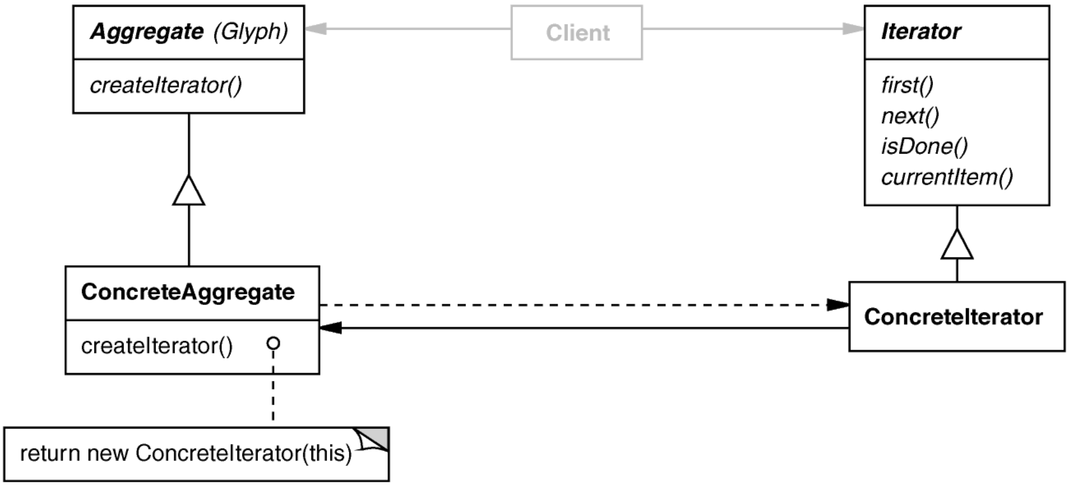
Intent

access elements of a container without exposing its representation

Applicability:

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Structure:



Consequences

+ flexibility: aggregate & traversal are independent.

+ multiple iterators & multiple traversal algorithms.

+ additional communication overhead between iterator & aggregate.

Implementation

- internal versus external iterators.
- violating the object structure's encapsulation.
- robust iterators .
- synchronization overhead in multi-threaded programs.
- batching in distributed & concurrent programs.

Known Uses

- C++ STL iterators.
- JDK Enumeration, Iterator .
- Unidraw iterator.

Visitor:

- defines action(s) at each step of traversal.
- avoids wiring action(s) into Glyphs.
- iterator calls glyph's accept(Visitor) at each node.
- accept() calls back on visitor (a form of "static polymorphism" based on method overloading by type).

```
void Character::accept (Visitor &v) { v.visit
(*this); } class Visitor {
public:
   virtual void visit (Character
   &);    virtual    void    visit
   (Rectangle &); virtual void
   visit (Row &);
   // etc. for all relevant Glyph subclasses
};
```

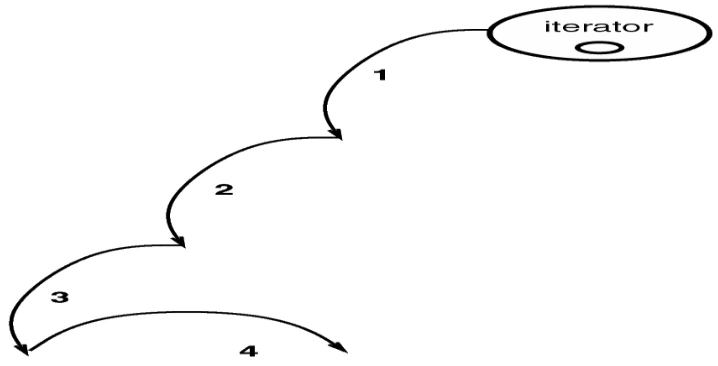
SpellingCheckerVisitor :

- gets character code from each character glyph.

  Can define getCharCode() operation just on Character() class
- checks words accumulated from character glyphs.
- combine with PreorderIterator .

```
class SpellCheckerVisitor : public
Visitor { public:
   virtual void visit (Character
   &);    virtual    void    visit
   (Rectangle &); virtual void
   visit (Row &);
   // etc. for all relevant Glyph
subclasses Private:
std::string accumulator_;
};
```

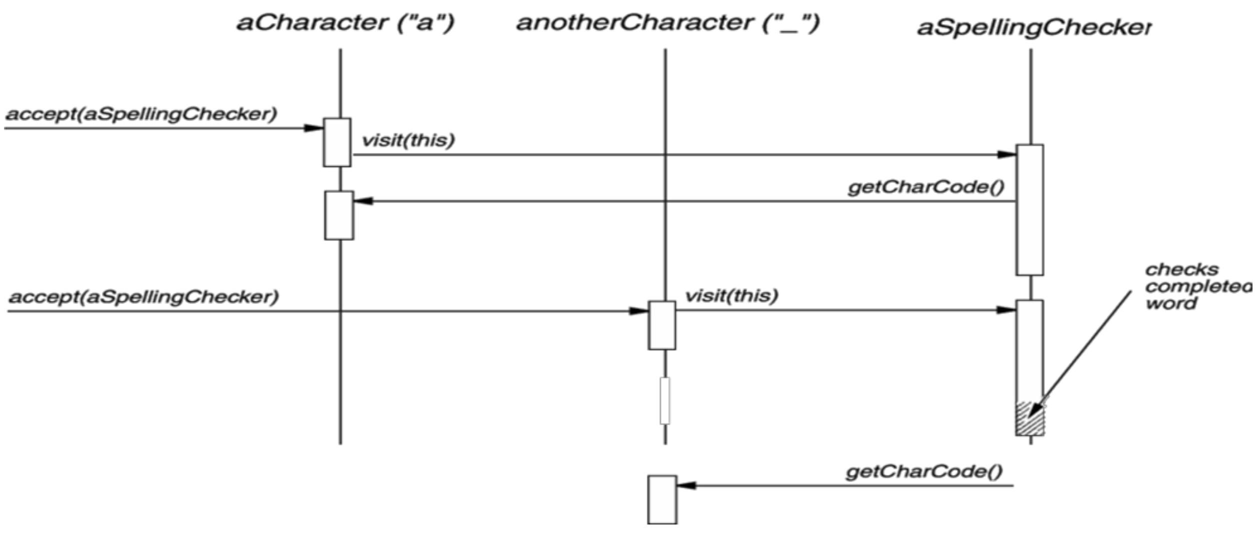
Accumulating Words:



Spelling check performed when a nonalphabetic character it reached.

Interaction Diagram:

- The iterator controls the order in which accept() is called on each glyph in the composition.
- accept() then "visits" the glyph to perform the desired action.
- The Visitor can be sub-classed to implement various desired actions.



HyphenationVisitor:

- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

```
class HyphenationVisitor : public
Visitor { public:
   void visit (Character
   &);      void      visit
   (Rectangle  &);  void
   visit (Row &);
   // etc. for all relevant Glyph subclasses
};
```

Concluding Remarks:

- design reuse.
- uniform design vocabulary.
- understanding, restructuring, & team communication.
- provides the basis for automation.
- a “new” way to think about design.

Creational Patterns :

- Abstracts instantiation process
- Makes system independent of how its objects are¬

– created

– composed

– represented

- Creational patterns encapsulates knowledge about which concrete classes the system uses
- Hides how instances of these classes are created and put together
- Important if systems evolve to depend more on object composition than on class inheritance
- Emphasis shifts from hardcoding fixed sets of behaviors towards a smaller set of composable fundamental behaviors
- Encapsulate knowledge about concrete classes a system¬ uses
- Hide how instances of classes are created and put together

What are creational patterns?

- Design patterns that deal with object creation¬ mechanisms, trying to create objects in a manner suitable to the situation
- Make a system independent of the way in which¬ objects are created, composed and represented

Recurring themes :

- Encapsulate knowledge about which concrete classes the system uses (so we can change them easily later)
- Hide how instances of these classes are created and put together (so we can change it easily later)

Benefits of creational patterns :

Creational patterns let you program to an interface defined by an abstract class that lets you configure a system with "product" objects that vary widely in structure and functionality

Example:

GUI systems.

Interviews GUI class

library. Multiple look-

and-feels.

Abstract Factories for different screen components.

Generic instantiation – Objects are instantiated¬ without having to identify a specific class type in client code (Abstract Factory, Factory) .

Simplicity – Make instantiation easier: callers do not¬ have to write long complex code to instantiate and set up an object (Builder, Prototype pattern).

Creation constraints – Creational patterns can put¬ bounds on who can create objects, how they are created, and when they are created .

Abstract Factory Pattern

Abstract factory provide an interface for creating families of related or dependent objects without specifying their concrete classes

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

Also Known As: Kit.

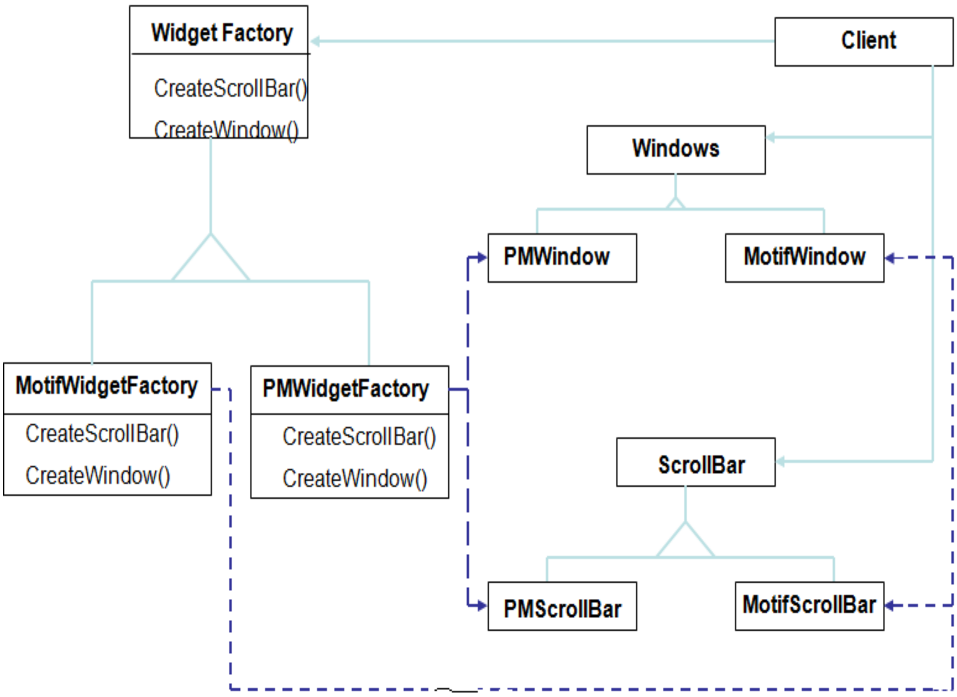
Motivation:

User interface toolkit supports multiple look-and-feel standards (Motif, Presentation Manager).

Different appearances and behaviors for UI

widgets Apps should not hard-code its widgets

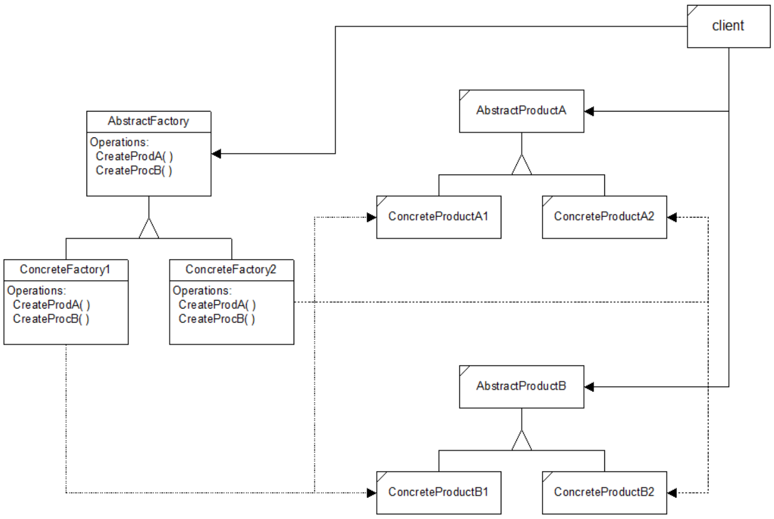
ABSTRACT FACTORY
Motivation



Solution:

- Abstract Widget Factory class
- Interfaces for creating each basic kind of widget
- Abstract class for each kind of widgets,
- Concrete classes implement specific look-and-feel.

Abstract Factory Structure



Abtract Factory :

Declares interface for operations that create abstract product objects

Concrete Factory :

– Implements operations to create concrete product objects

Abstract Product :

– Declares an interface for a type of product object.

Concrete Product:

– Defines a product object to be created by concrete factory

– Implements the abstract product interface

Client:

– Uses only interfaces declared by Abstract Factory and AbstractProduct classes.

Collaborators :

- Usually only one ConcreteFactory instance is used for an activation, matched to a specific application context. It builds a specific product family for client use -- the
  client doesn’t care which family is used -- it simply needs the services appropriate for the current context.

- The client may use the AbstractFactory interface to initiate creation, or some other agent may use the AbstractFactory on the client’s behalf.

Presentation Remark :

- Here, we often use a sequence diagram (event-trace) to show the dynamic interactions between participants.

- For the Abstract Factory Pattern, the dynamic interaction is simple, and a sequence diagram would not add much new information.

Consequences :

- The Abstract Factory Pattern has the following benefits:
  - It isolates concrete classes from the client.
    - You use the Abstract Factory to control the classes of objects the client creates.
    - Product names are isolated in the implementation of the ConcreteFactory, clients use the instances through their abstract interfaces.
  - Exchanging product families is easy.
    - None of the client code breaks because the abstract interfaces don’t change.
    - Because the abstract factory creates a complete family of products, the whole product family changes when the concrete factory is changed.

- It promotes consistency among products.
    - It is the concrete factory's job to make sure that the right products are used together.

More benefits of the Abstract Factory Pattern

- It supports the imposition of constraints on product families, e.g., always use A1 and B1 together, otherwise use A2 and B2 together.

- **The Abstract Factory pattern has the following liability:**
    - Adding new kinds of products to existing factory is difficult.

- Adding a new product requires extending the abstract interface which implies that all of its derived concrete classes also must change.
- Essentially everything must change to support and use the new product family
- abstract factory interface is extended
- derived concrete factories must implement the extensions
- a new abstract product class is added
- a new product implementation is added
- client has to be extended to use the new product

Implementation

- Concrete factories are often implemented as singletons.
- Creating the products
    - Concrete factory usually use the factory method.
        - simple
        - new concrete factory is required for each product family
    - alternately concrete factory can be implemented using prototype.
        - only one is needed for all families of products

- product classes now have special requirements - they participate in the creation

- Defining extensible factories by using create function with an argument
  - only one virtual create function is needed for the AbstractFactory interface
  - all products created by a factory must have the same base class or be able to be safely coerced to a given type
  - it is difficult to implement subclass specific operations

Know Uses:-

- **Interviews**
  - used to generate "look and feel" for specific user interface objects
  - uses the Kit suffix to denote AbstractFactory classes, e.g., WidgetKit and DialogKit.
  - also includes a layoutKit that generates different composite objects depending on the needs of the current context

ET++

- another windowing library that uses the AbstractFactory to achieve portability across different window systems (X Windows and SunView).

Related Patterns:-

- Factory Method -- a "virtual" constructor
- Prototype -- asks products to clone themselves
- Singleton -- allows creation of only a single instance

Code Examples:-

- **Skeleton Example**
  - Abstract Factory Structure
  - Skeleton Code
- **Neural Net Example**
  - Neural Net Physical Structure
  - Neural Net Logical Structure
  - Simulated Neural Net Example

BUILDER :-

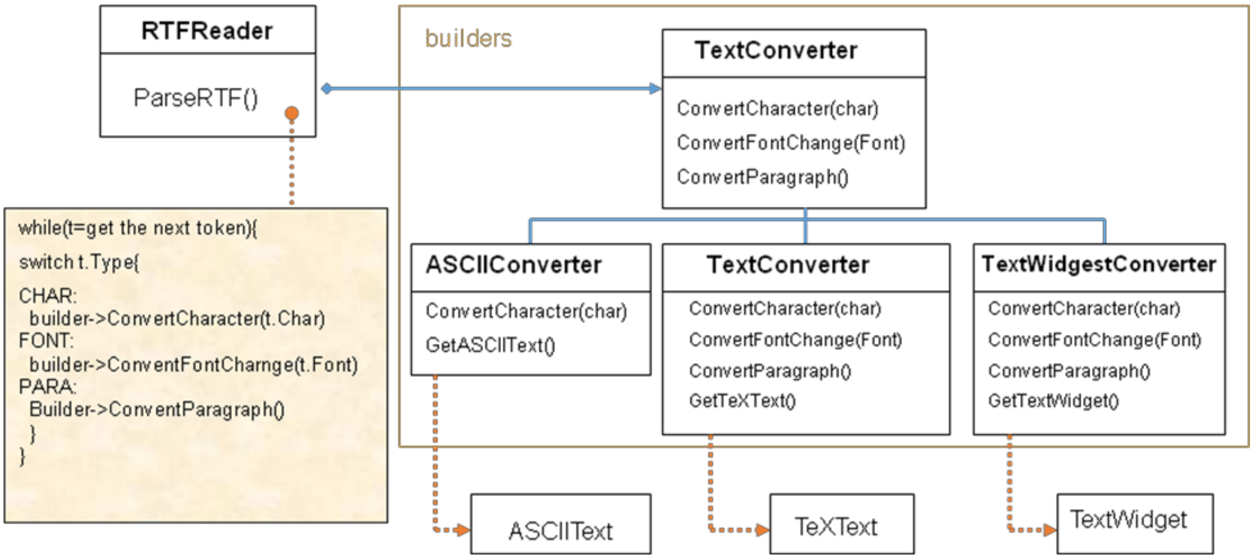
- **Intent:**

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

- Motivation:
- RTF reader should be able to convert RTF to many text format
- Adding new conversions without modifying the reader should be easy

- **Solution:**
  - Configure RTFReader class with a Text Converter object
  - Subclasses of Text Converter specialize in different conversions and formats
  - TextWidgetConverter will produce a complex UI object and lets the user see and edit the text

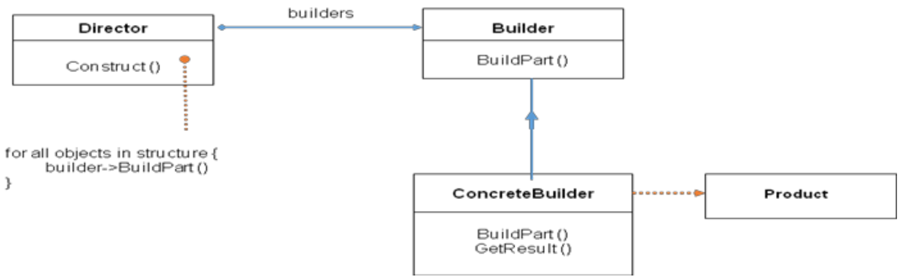
BUILDER Motivation:-



Applicability:-

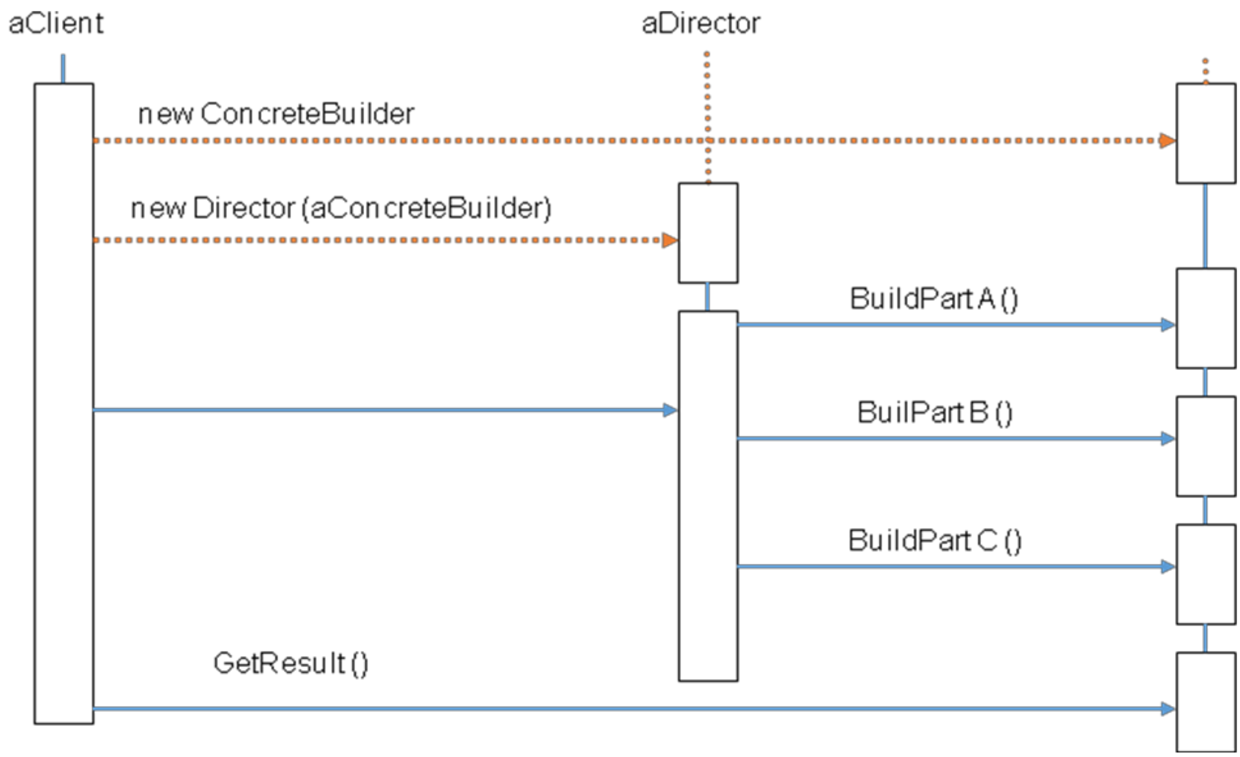
- Use the Builder pattern when
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - The construction process must allow different representations for the object that is constructed

BUILDER Structure:-



Builder – Collaborations:-

- Client creates Director object and configures it with the desired Builder object
- Director notifies Builder whenever a part of the product should be built
- Builder handles requests from the Director and adds parts to the product
- Client retrieves the product from the Builder



Why do we use Builder?

- Common manner to Create an Instance
  - *Constructor!*
  - Each Parts determined by Parameter of the Constructor

```
public class Room {
    private int area;
    private int windows;
    public String purpose;

    Room() {
    }

    Room(int newArea, int newWindows, String newPurpose){
        area = newArea;
        windows = newWindows;
        purpose = newPurpose;
    }
}
```

There are Only 2 different ways to Create an Instance part-by-part.

- **In the previous example,**
  - We can either determine all the arguments or determine nothing and just construct. We can't determine arguments partially.
  - We can't control whole process to Create an instance.
  - Restriction of ways to Create an Object
  - Bad Abstraction & Flexibility

Discussion:-

- Uses Of Builder
  - Parsing Program(RTF converter)
  - GUI

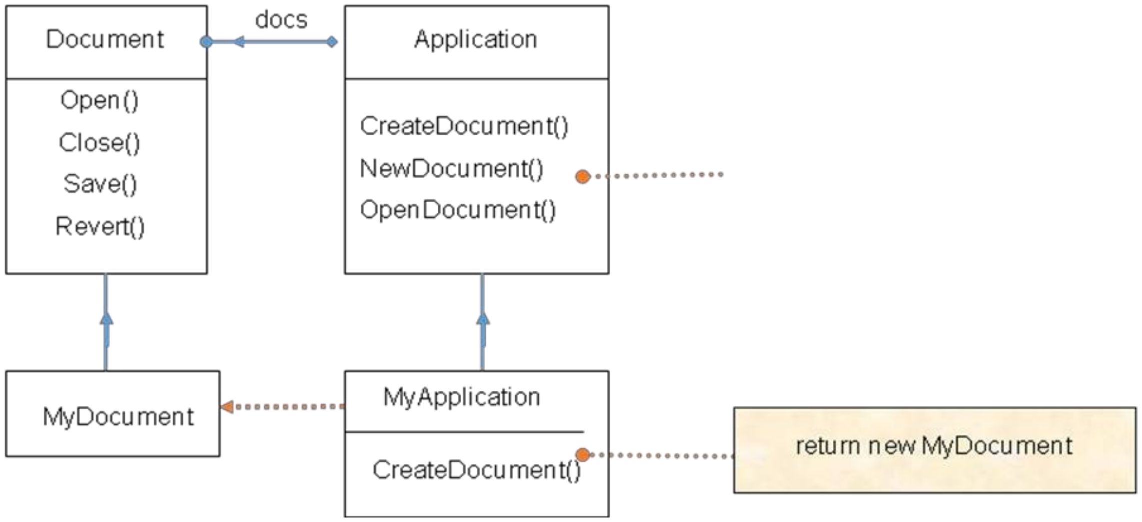
FACTORY METHOD (Class Creational):-

- **Intent:**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to subclasses.
- **Motivation:**
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate

Motivation:-

- Motivation: Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework
- Also Known As: Virtual Constructor

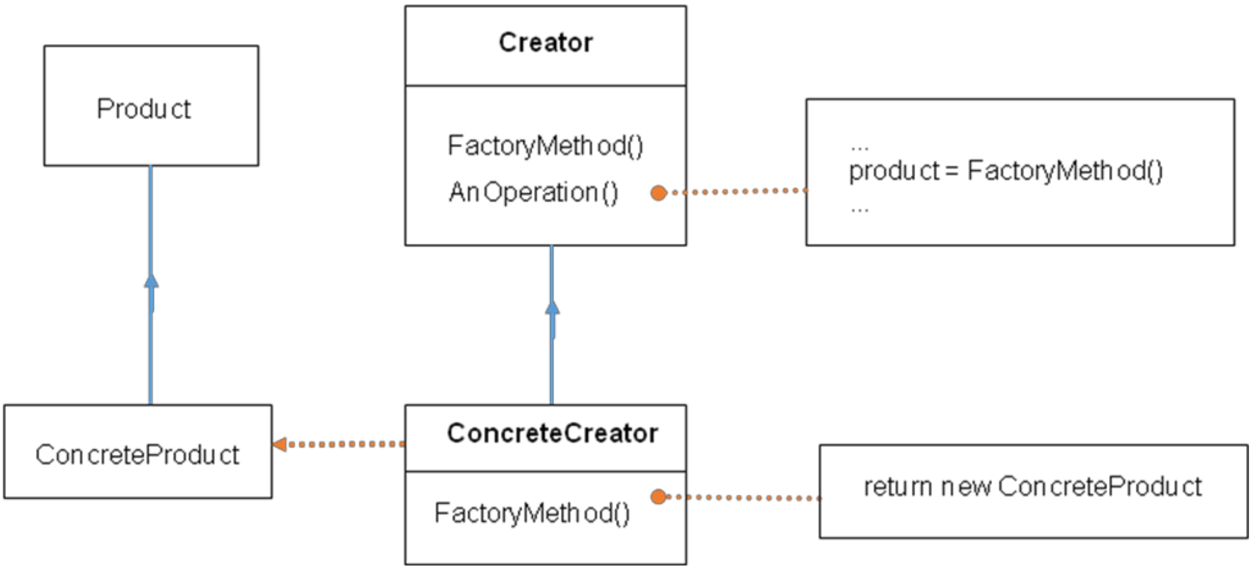
FACTORY METHOD Motivation:-



Applicability:-

- Use the Factory Method pattern when
  - 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.

FACTORY METHOD Structure:-



Participants:-

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type product

– May contain a default implementation of the factory method

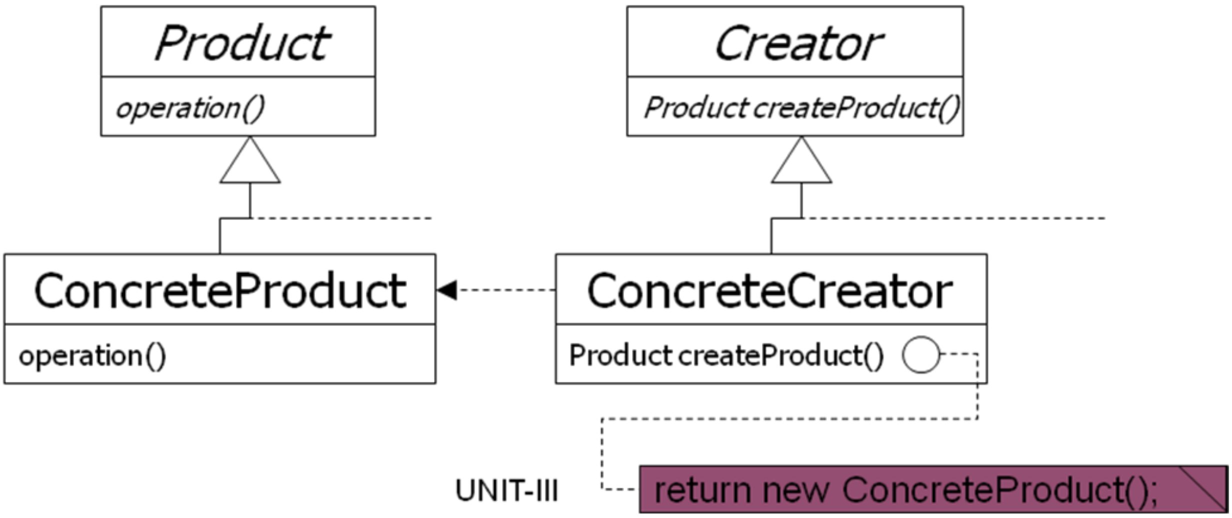
– Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.

- ConcreteCreator

  – Overrides factory method to return instance of ConcreteProduct

Factory Method:-

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- A related pattern is AbstractFactory



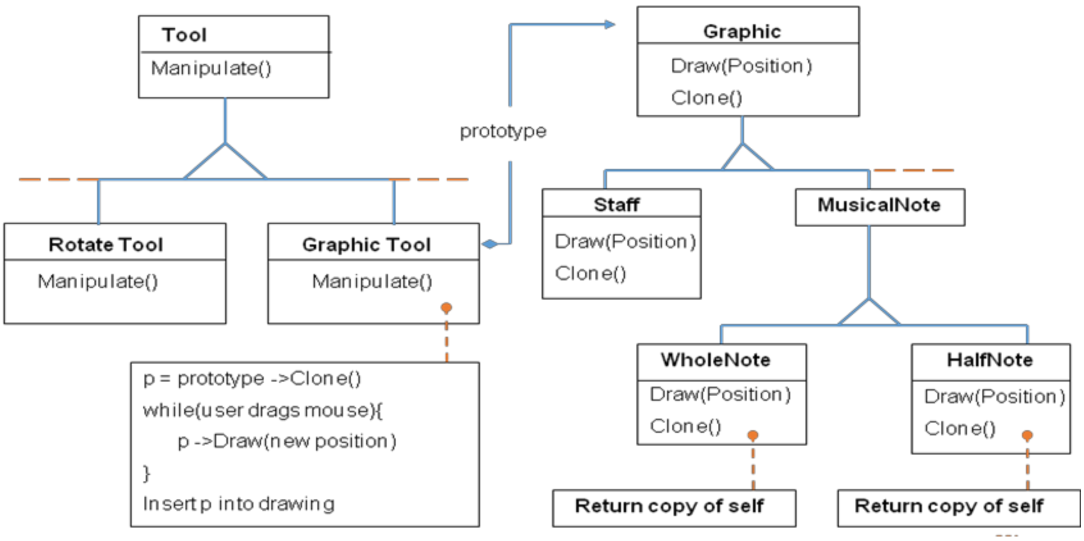
PROTOTYPE (Object Creational):-

- **Intent:**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Motivation:**
  - Framework implements Graphic class for graphical components and GraphicTool class for tools manipulating/creating those components

Motivation:-

- Actual graphical components are application-specific
- How to parameterize instances of Graphic Tool class with type of objects to create?
- Solution: create new objects in Graphic Tool by cloning a **prototype** object instance

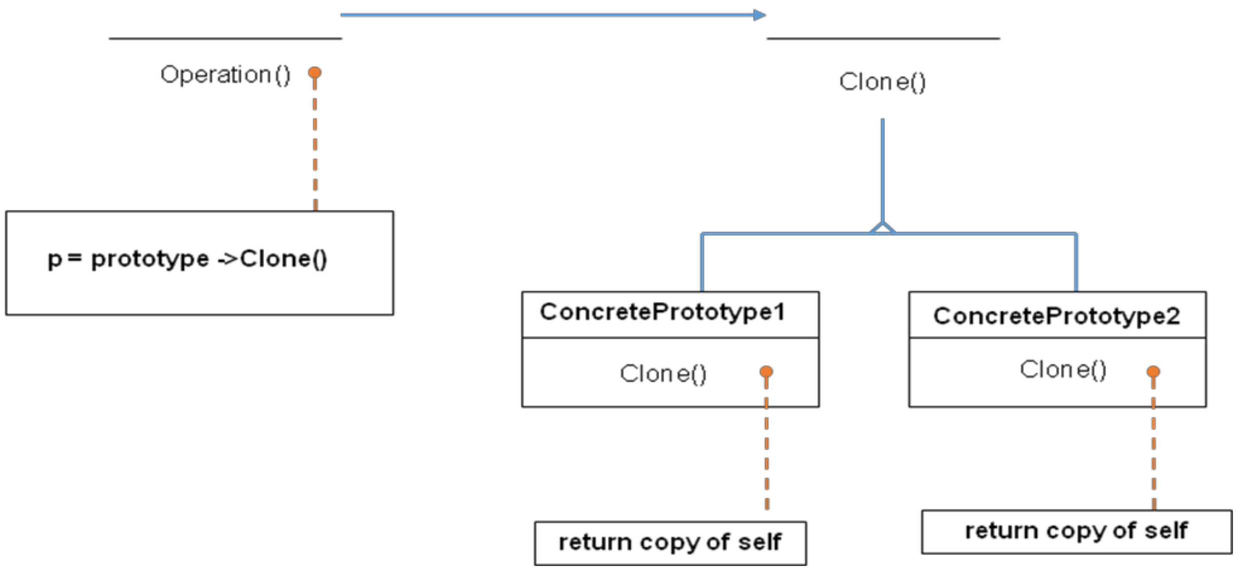
PROTOTYPE Motivation:-



Applicability:-

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

PROTOTYPE Structure:-



Participants:

- Prototype (Graphic)
  - Declares an interface for cloning itself
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a prototype to clone

itself Collaborations:

- A client asks a prototype to clone Itself.

SINGELTON:-

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation:
  - Some classes should have exactly one instance (one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but doesn't prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface

Applicability:-

- Use the Singleton pattern when
  - 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.