Unit - I

Chapter - 2 Case Study (Lexi Document Design)

Chapter - 3 Creational patterns.

Case Study (Lexi's Document Design editor).

define Lexi :-

The design of a "what-you-see-is-what-you-Get" (or WYSIWYG) document editor called Lexi.

✓ We can see the figure (Lexi's User Interface) page no: 34 (2.1).

✓ A WYSIWYG representation of the document occupies the large rectangular area in the center.

✓ The document can mix text and graphics freely in a variety of formatting styles.

✓ Observe the document are generally surrounded by pull-down menus, and scroll-bars, plus a collection of page icons for Jumping to a particular page in the document.

⇒ List out the How many problem examine while designing Lexi document editor.

examine 7 problems in lexi's design.

1. Document Structure

choice of internal representation for the document effects every aspect of Lexi's design.

- All editing, formatting, displaying, and textual analysis will require traversing the representation.
- The way we organizing this information will impact the design of the rest of the application. DP: Composite

2 Formatting

In this case we study the, How does lexi's designe text and graphics into lines and columns?

Strategy

✓ which objects are responsible for carry out different formatting policies.. and

study the policies, How interact with document's internal representation.

Ex: Increase font size, either adding or remove the image from docum

.so on.

3. Embellishing the User Interface—

Embellishing ~~anything~~ can mean, make (something) more attractive by the addition of decorative details or features.

my ref followers often embellish stories about their heros / blue silk embellished with golden embroidery.

ex: change the font style / ~~language~~ DP: Decorator

4. Supporting multiple look-and-feel Standards

ex: Layout ~~that~~ of document.

Lexi should adapt easily to different look-and-feel standards such as motif and Presentation Manager (PM) without major modification. DP: Abstract factory

5. Supporting multiple window systems.

Different look-and-feel standard are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible.

ex: Supporting multiple

DP: Bridge

3. Embellishing the User Interface —

Embellishing ~~gathering~~ can mean, ~~thesaurus~~ make (something) more attractive by the addition of decorative details or features.

my ref

followers often embellish stories about their heroes / blue silk embellished with golden embroidery.

ex: change the font style / ~~language~~

DP: Decorator

4. Supporting multiple look-and-feel Standards

ex:- Layout ~~that of~~ document.

Lexi should adapt easily to different look-and-feel standards such as motif and Presentation Manager (PM) without major modifications.

DP: Abstract factory

5. Supporting multiple window systems.

Different look-and-feel standard are usually implemented on different window systems.

Lexi's design should be as independent of the window system as possible.

ex: Supporting multiple

DP: Bridge

6. User operation.

✓ Users control Lexi through various user interfaces, including buttons and pull-down menus.

✓ The functionality behind those interfaces is scattered throughout the objects in the application.

✓ Here is to provide a uniform mechanism both for accessing this scattered functionality and for undoing its effects.

ex: mouse operation & button operation.

DP: Command

7 Spell checking & hyphenation-

hyphenation mean to use a hyphen to join two words or two parts of words or splitting of word

How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points?

✓ How can we minimize the number of classes we have to modify to add a new analytical operation?

Iterator

⇒ Define document & explain implementation of Document Structure in Lexi's.

document is an arrangement of basic graphical elements such as like characters, lines, polygons, figures, tables and other substructures.

✓ all the above element capture the total information content of the document.

Lexi's users manipulate substructures directly.

ex: ✓ user should treat a diagram as unit rather than a collection of individual elements

✓ user refer to a table as a whole, not as an unstructured mass of text and graphics.

It helps make the interface simple and intuitive. To give Lexi's implementation similar qualities, we call chose an internal representation that matches the document's physical structure.

The interal representation should support the following:

✓ maintain the document's physical structure i.e, the arrangement of text and graphics into lines, columns, tables, etc

✓ Generating and presenting the document visually.

✓ Mapping positions on the display to elements in the internal representation. This lets Lexi determine what the user reffering to when he points to something in the visual representation.

In addition to these goals are some constraints

✓ treat the text and graphics uniformly mean that avoid treating graphics as a special case of text or text as a special case of graphics. we use one set of mechanism for both text and graphics

✓ in implementation should n't have to distinguish b/w single elements and groups of elements in the internal representation.

* Lexi should be able to treat simple and complex elements uniformly thereby allowing arbitarily complex documents

for that we apply use composite design pattern

✓ we use a technique 'recursive composition' to represent hierarchically structured infor'n.

✓ Recursive composition gives a way to compose a document out of simple graphical elements.

as a 1st step,
we can tile a set of characters and graphics from left to right to form a line in the document.

2. multiple lines can be arranged to form

a c

PPT

Unit-II

Part B Creational patterns:

In S/W engineering

- Creational design patterns consider the object creation mechanism trying to create objects in a manner suitable to the situation.

- The basic form of object creation could result in design problems or in added complexity to design.

ex:- generally in Java/C++

Student stu = new Student();

- creational design patterns solve this problem by controlling the object creation.

- Creational design patterns are composed of two dominant ideas

1. encapsulating knowledge about which concrete classes the system uses.

2. hiding how instances of these concrete classes are created and combined.

✓ creational design patterns categorized into object creational patterns & class-

there
✓ object creational patterns deal with object creation, and class creational patterns deal with class instatiation.

✓ object creational patterns defer part of its object creation to another object, while class creational pattern defer its objects creation to subclasses.

Creational

Definition

The creational patterns aim to separate a system from how its objects are created, composed, and represented.

They increase the system's flexibility in terms of the what, who, how, and when object creation.

- usage

✓ modern s/w engg depends more on object composition than class inheritance; emphasis shifts away from hard coding. behaviors towards defining a smaller set of basic behaviors that can be composed into more complex ones.

→ Hard-coding behaviors are inflexible because they require overriding / re-impl'ng the whole thing in order to change parts of design.

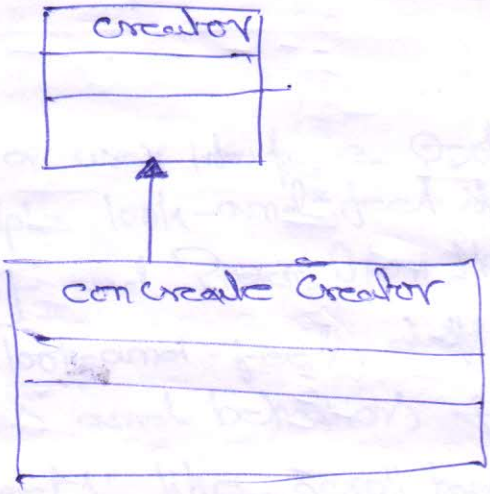
- ✓ hard-coding doesn't promote reuse and makes it difficult to keep track of errors.
- For this creational patterns are more useful than hard-coding behaviors.
- ✓ Creational patterns makes design more flexible.
- They provide different ways to remove explicit references in the concrete classes from the code that needs to instantiate them.

✓ * Applying creational patterns (when)

- A system should be independent of how its objects and products are created.
- A set of related objects is designed to be together.
- Hiding the implementations of class library or product, revealing only their interfaces.
- Constructing different representation of independent complex objects.
- A class wants its subclass to implement the object it creates.
- The class instantiations are specified at run-time.

- There must be a single instance and client can access this instance at all times.
- instance ~~should be~~ should be extensible without being modified.

Structure



Example

- Abstract Factory — class creational pattern.
- Factory Method
- Builder
- Prototype
- Singleton

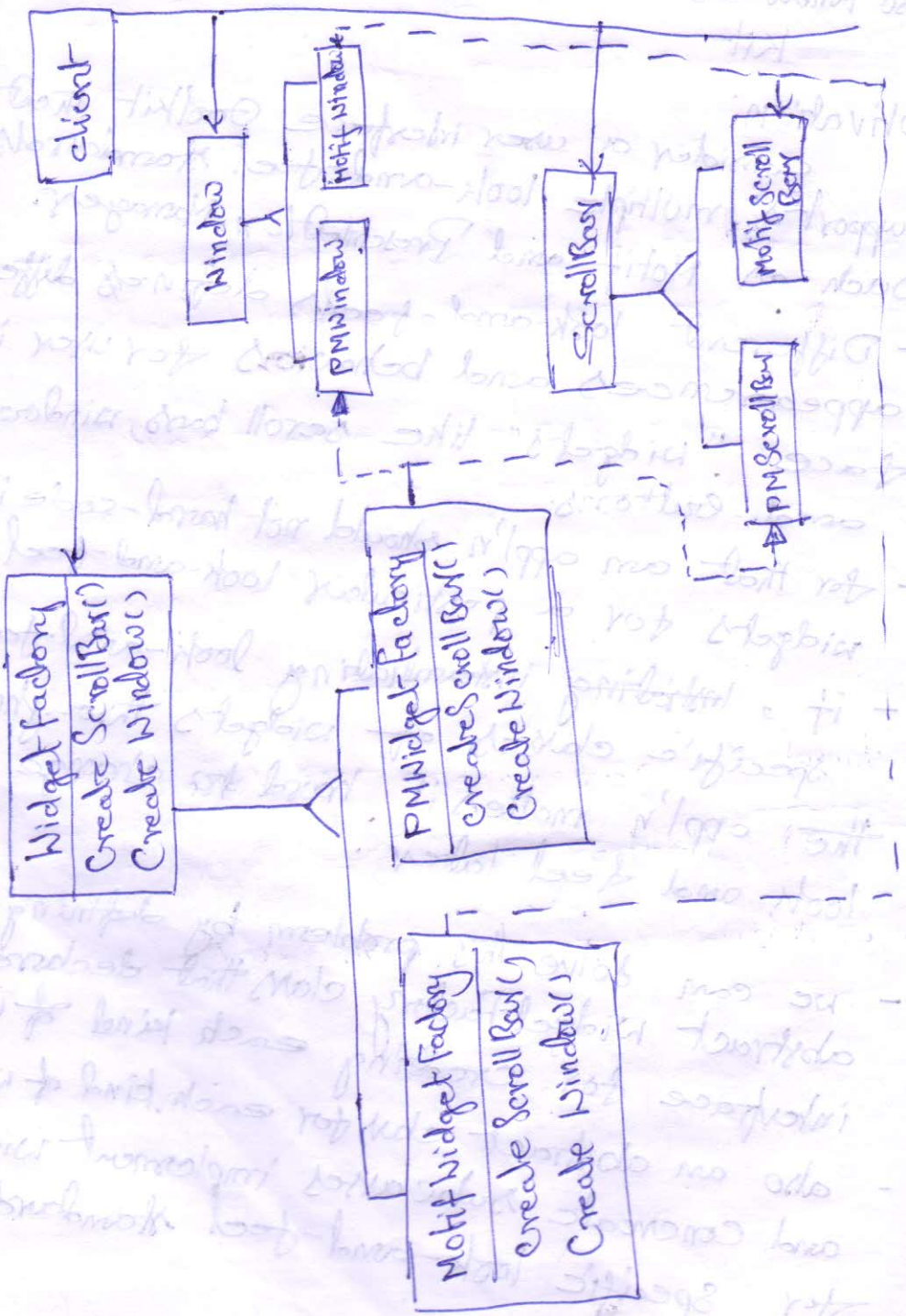
Abstract Factory

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

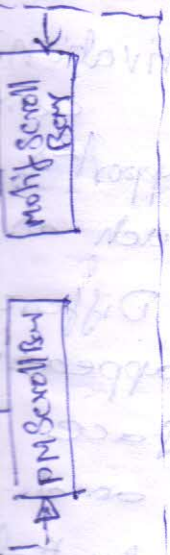
Also know As
Kit

Motivation:
consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager.
- Different look-and-feels defines different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.
- for that an appl'n should not hard-code its widgets for a particular look-and-feel.
- it, instantiating implementing look-and-feel-specific classes of widgets throughout the appl'n makes it hard to change the look and feel later.
- we can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each kind of widget.
- also an abstract class for each kind of widget and concreate subclasses implement widgets for specific look-and-feel standards.

- end users/clients call these operations to obtain widget instances, but they aren't aware of the concrete classes they're using.
- to clients stay indepent of the prevailing look-and-feel.



operations but they aren't they're using. prevailing look-



Applicability:

Use abstract factory in following cases

- a system should be independent of how its products are created, composed, and represented
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products,

Structure:

refer WidgetFactory - AbstractFactory (interface)
<type> WidgetFactory - Concrete factory's
<type> Product A/B - Abstract product-A/B
<type> window/scroll bars - concrete class.

Participants

Abstract factory
- declares an interface for operations that create abstract product objects

AbstractProduct
- declares an interface for a type of product object.

- ConcreteFactory - implements the operations to create concrete product objects.
- ConcreteProduct
  defines a product objects to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface

client
- uses only interfaces declared by abstractfactory & Abstract Product classes.

Collaborations:

a single instance of a Concretefactory class is created at run-time. This ~~creat~~ concrete factory creates product objects having a particular implementation.

- To create different product objects, client should use a different concrete factory.

- Abstractfactory defers creation of product objects to its concretefactory subclass.

Consequences:

✓ it isolates concrete classes
✓ It makes exchanging product families easy.
✓ It provide consistency among products.
✓ Supporting new kinds of products is difficult

Known uses:

Related Patterns:-

Abstract factory classes are often implemented with factory methods, but can also be implemented using prototype.

✓ singleton.

ex: buttons and checkboxes will act as products. They have two variants: Mac OS & Windows.

✓ abstract factory defines an interface for creation buttons and checkboxes.

✓ for that two concrete factories, which return both products in a single variant.

+ client code works with factories and products using abstract interfaces. it makes the same client code working with many product variants, depending on the type of factory object.

Builder

Intent

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

Motivation :

, A reader for the RTF document exchange format should be able to convert RTF to many text formats.
ex: plain ASCII text / text widget / MS-word.

✓ for that here we configure the RTFReader class with a TextConverter object that converts RTF to another textual representation.

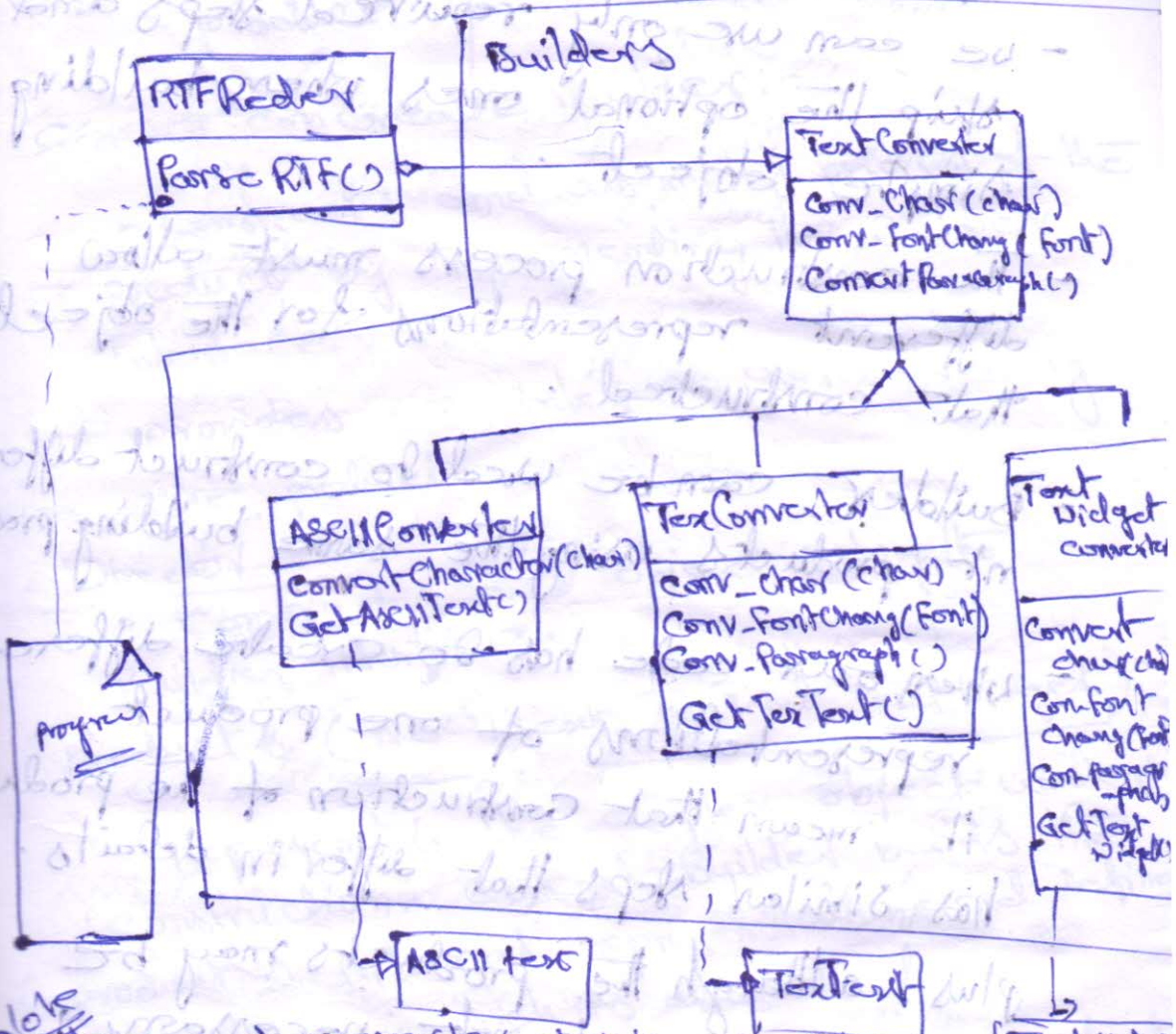
✓ The RTF reader parses the RTF document, it uses the TextConverter to perform the conversion.

✓ whenever the RTFReader recognizes an RTF token, it issues a request to the TextConverter to convert the token.

✓ TextConverter objects are responsible both for performing the data conversion and for representing the token in

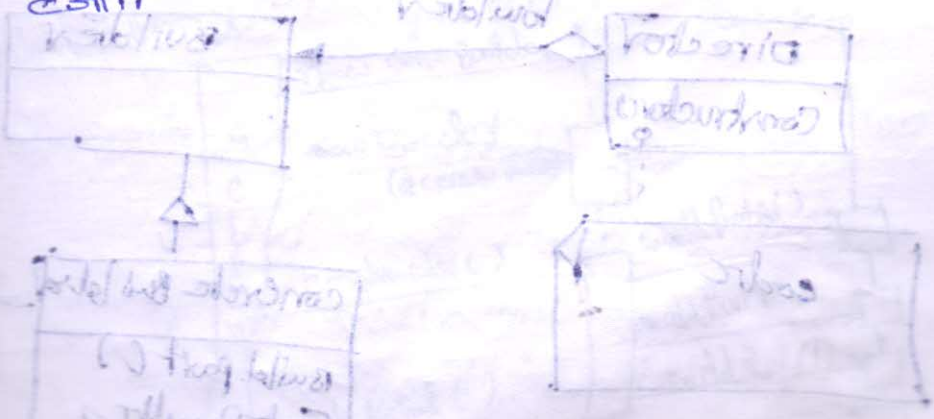
a particular format.

Subclasses of TextConverter specialize in different conversions and formats.



Here each converter class is a builder. The possible conversions are represented.

Example – construction of building by contractors / own:

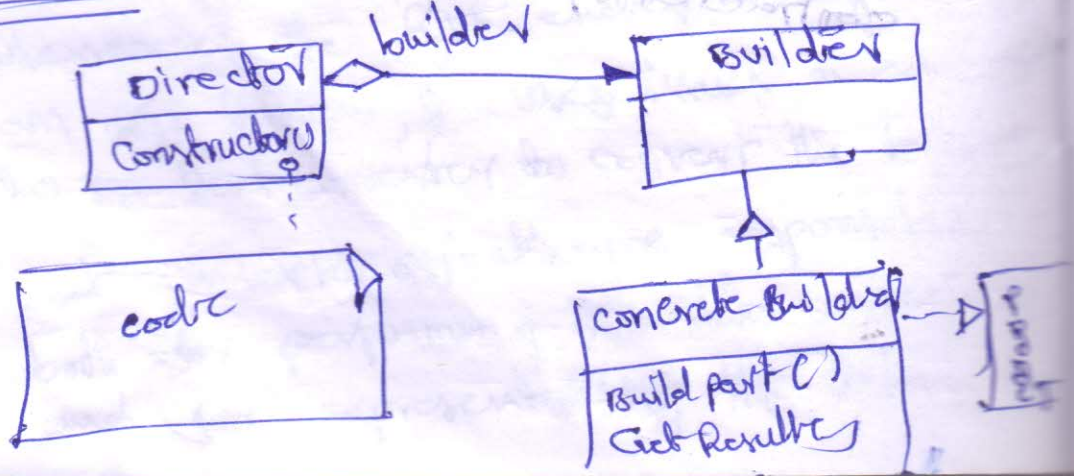


Applicability:

The builder pattern allows building object step by step.

- we can use only required steps and skip the optional ones when building a simple object.
- The construction process must allow different representations for the object that constructed.
- Builder can be used to construct differe nt products using the same building proces.
- When our code has to create different representations of one product
- it mean that construction of the product has similar steps that differ in details.
- plus, although the products may be similar, they do not necessary have to have a common base class or interface.

Structure



Participant:

Builder → Specifies an abstract interface for creating parts of a Product object

ConcreteBuilder

Director

Product.

ConcreteBuilder:—

- constructs and assembles parts of the product by implementing the Builder interface.
- provides an interface for retrieving the product.

Director

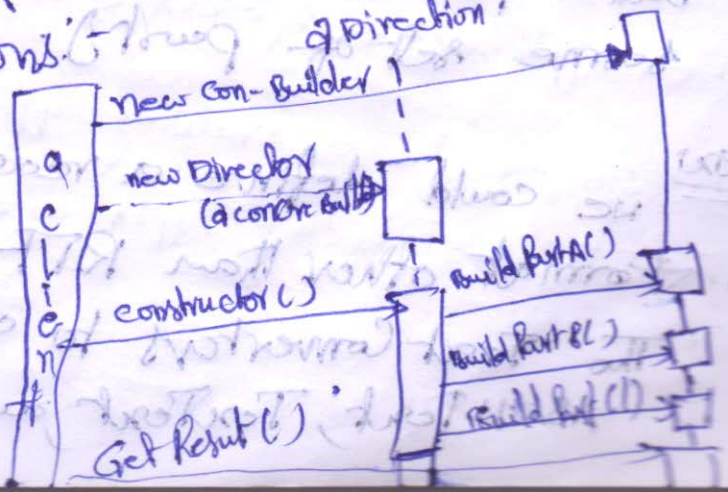
constructs and object using the Builder interface.

Products (ASCIIText, TeXText, TextWidget)

Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.

It includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations:-



Consequences:

Allows building products step by step

Allows Using the same code for building different products.

- Isolates the complex construction code from a product's core business logic.
- Increases overall code complexity by creating multiple additional classes.

* ~~easy to achieve~~

~~Vary a product internal representation~~

✓ Vary the product internal representation very easy with Builders.

✓ It isolates codes for construction and representation.

- Each concrete Builder contains all the code to create and assemble a particular kind of product,
- different Directors can reuse it to build product variants from the same set of parts.

Ex: we could define a reader for a format other than RTF,

The TextConverters to generate ASCIIText, TeXText, and Text

Factory Method

Intent:

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Also known As:

Virtual constructor

Motivation

Frameworks use abstract classes to define and maintain relationship b/w objects and a frame work is often responsible for creating these objects as well.

- consider a frame work for applications that can present multiple documents to the user.
- two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application specific implementations.

ex: To create a drawing appl'n, we define the classes DrawingAppl'n and Drawing Document.

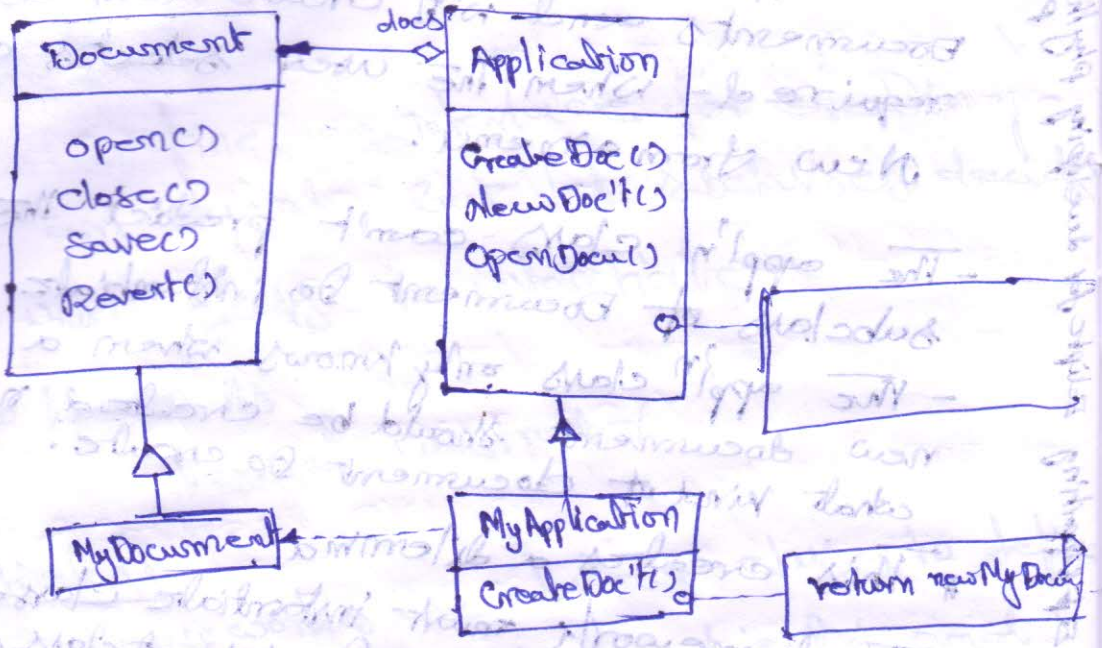
- The appl'n class responsible for managing Documents and with create them as required - When the user selects open/ New from a menu.

- The appl'n class can't predict the subclass of Document to instantiate.
- The appl'n class only knows when a new document should be created, not what kind of document to create.

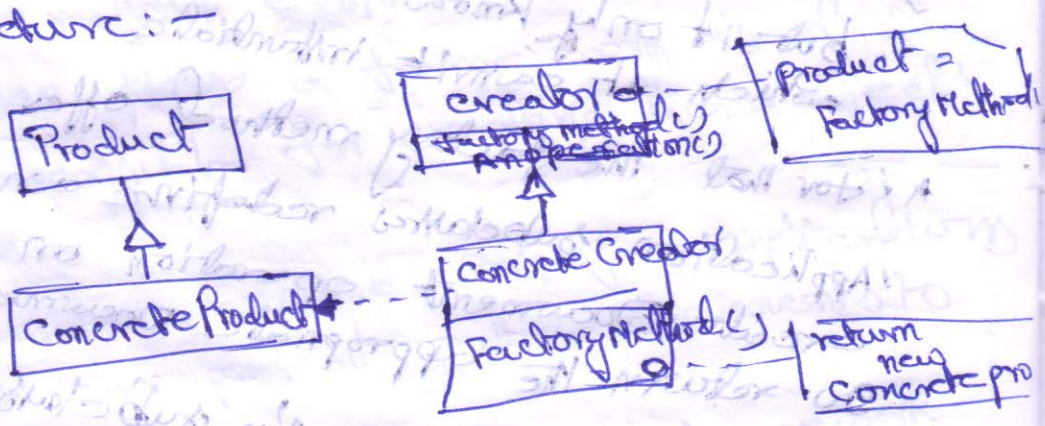
* This creates a dilemma:
The framework must instantiate classes but it only know about abstract classes, which it can't instantiate.

* / for that the factory method offer a solution.
Application subclasses redefine an abstract create Document operation on Appl'n to return the appropriate Document subclass.

- once an Appl'n subclass instantiate d, it can then instantiate application-specific documents without knowing their class. Here we call createDocument a factory method because it's responsible for "manufacturing" an object.



Structure :-



Participants.

Product
defines the interface of objects the factory method creates

ConcreteProduct —
implements the product interface.

Creator —
declares the factory method, which returns an object of type product.
- Creator may also define a default impl' of the factory method that returns a

default
Concrete
overri
return a
Collaboration
Creator
define
- so that
appropri
Consequenc
Factory
bind
code
- ⁂ fo
- disadv
cl
class
product
- provid
cr
with a
flexible
factory
providin

default concrete product.

Concrete Creator:

overrides the factory method to return an instance of a concrete product.

Collaborations

- Creator relies on its subclasses to define the factory method.
- so that, it returns an instance of the appropriate concrete product.

Consequences

Factory methods eliminate the need to bind appln specific classes into your code.

- factory method can work with any user defined concrete Product classes.

- disadvantage

client must subclass the creator class to create a particular concrete product object.

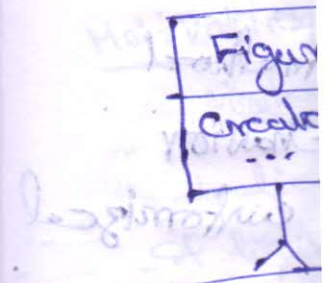
- provide hooks for subclasses — creating objects inside a class with a factory method is always more flexible than creating an object directly. factory method gives subclasses a hook for providing an extended version of an object.

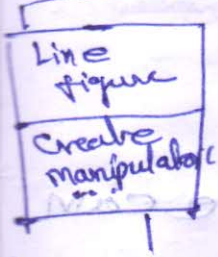
- Connects parallel class hierarchies:-
  clients can find factory methods useful, especially in the case of parallel class hierarchies.

* Parallel class hierarchies result when a class delegates some of its responsibilities to separate class.
* Consider graphical figures that can be manipulated interactively, that is, they can be stretched, moved, or rotated using the mouse.
- for that requires storing and updating of information, that records the state of the manipulation at a given time.
  this state is needed only during manipulation;
- moreover, different figures behaved differently when the user manipulates them.

Ex: stretching a line figure might have the effect of moving an endpoint, where as stretching a text figure may change its line spacing.

- with these constraints, to use a separate manipulator object that

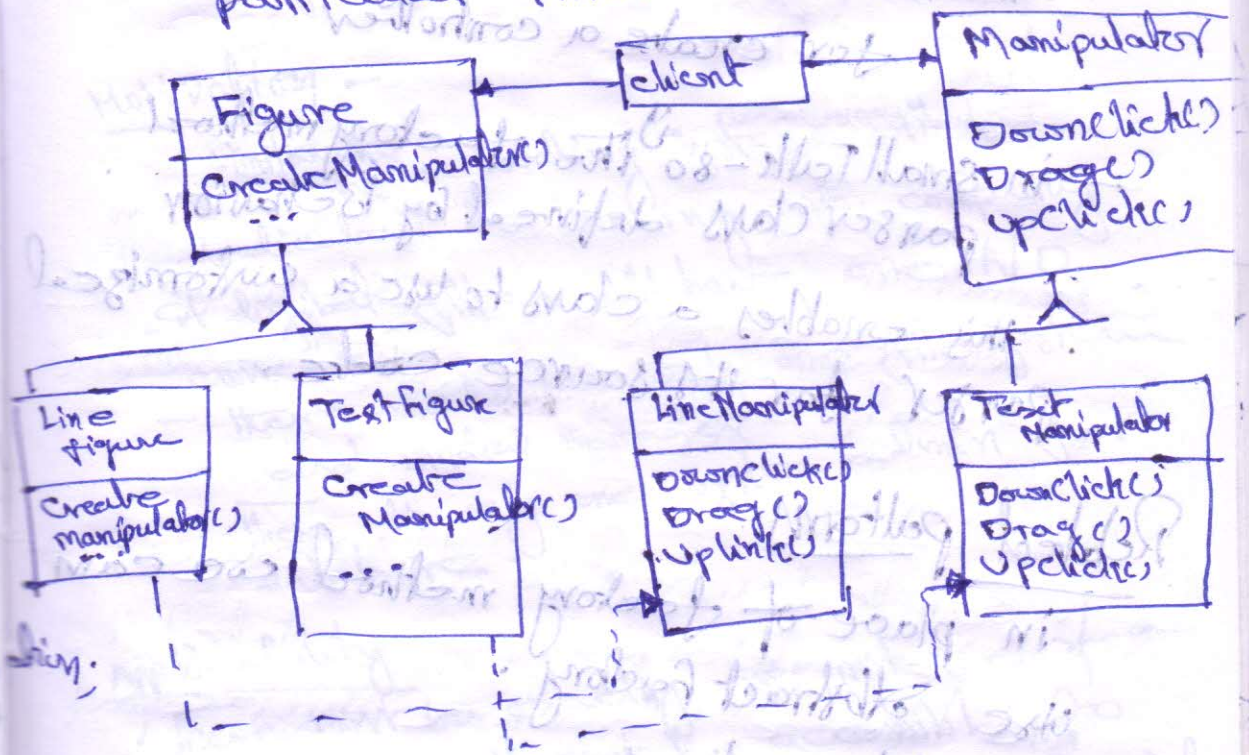




Implementa
- Paramet
- Langua
- using
- Naming
Sample co

implements the interaction and keeps track of any manipulation-specific state that is needed.

- Different figures will use different manipulator subclasses to handle particular interactions.



Implementation

- Parameterized factory method
- Language-specific variants and issues
- using templates to avoid subclassing
- Naming Conventions.

Sample code

Known uses:
factory methods prevade toolkits and frameworks.
- In MVC for implementing default controller for create a controller
- in SmallTalk-80 the factory method parserClass defined by Behavior
this enables a class to use a customized parser for its source code.

Related patterns:
in place of factory method, we can use Abstract Factory.
- template method.
- Prototype.

Singleton pattern

Intent

Singleton pattern Ensure a class only has one instance, and provide a global point of access to it.

Motivation –

~~similar to~~ An accounting system will be dedicated to serving one company.
- A digital filter will have one A/D converter.
- There should be only one file system and window manager.
- There is only one system admin for database.

Applicability –

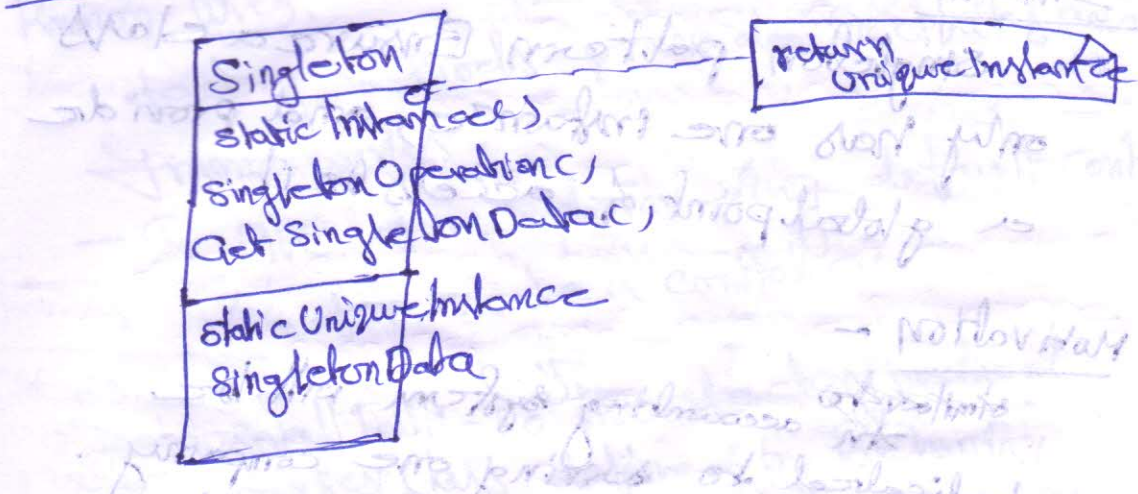
✓ There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

✓ When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance.
  - Instance is a class operation.
  - ex: static member function in C++
  - it responsible for creating its own unique instance.

Structure:



Collaborations

- clients access a Singleton instance through Singleton's Instance operation.

Consequences:

✓ controlled access to sole instance:
The singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

✓ Reduced name space:
The singleton pattern is an improvement over global variables.
- it avoids polluting the name space with global variables that store sole instances

✓ Permits refinement of operations and representation
The singleton class may be subclassed and its easy to configure an application with an instance of ~~this singleton class~~ this extended class

we can configure the application with an instance of the class it need at run-time.

✓ Permits a variable number of instances - the pattern makes it easy to change your mind and allow more than one instance of the singleton class.

moreover, we can use the same approach to control the number of instances that the application uses.

Here the operation grants access to the singleton instance needs to change

✓ More flexible than class operations -

Simple mentation

Ensuring a Unique instance -

- The singleton pattern makes the sole instance a normal instance of a class.
- for that we hide the operation that create the instance behind a class operation (that is, either a static member function / a class method) that guarantees only one instance is created.

- This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value.
- This approach ensures that a Singleton is created and initialized before it's first use.

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};

impl'n  Singleton* Singleton::_instance = 0;
        Singleton* Singleton::Instance() {
            if (_instance == 0)
            {
                _instance = new Singleton;
            }
            return _instance;
        }
```

note: It is not enough to define the singleton as a global or static object and then rely on automatic initialization.

Creational Patterns: continuation

Singleton

Implementation:

2. Subclassing the singleton class:

known uses

- Relationship b/w classes & their metaclasses
  a metaclass is the class of a class,
  and each metaclass has one instance.
  metaclasses don't have names, but they
  keep track of their sole instance
  and will not normally create another
- The Interview user interface toolkit uses
  the singleton pattern to access the
  unique instance of its session and
  Widgetkit classes among others.

Related patterns

many patterns can be implemented using the singleton pattern.

Abstract Factory, Builder, and Prototype.

Prototype Pattern

Intent

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

Motivation:

Applicability

Use the prototype pattern when a system should be independent of how its products are created, composed, and represented.

- ✓ to avoid building a class hierarchy of factories that parallels the class hierarchy of products.
- ✓ when instances of a class can have one of only a few different combinations of state.

Structure

client

Operati

P = Proto

Participants

Prototype

decla

Concrete Pro

imp

cloning

client

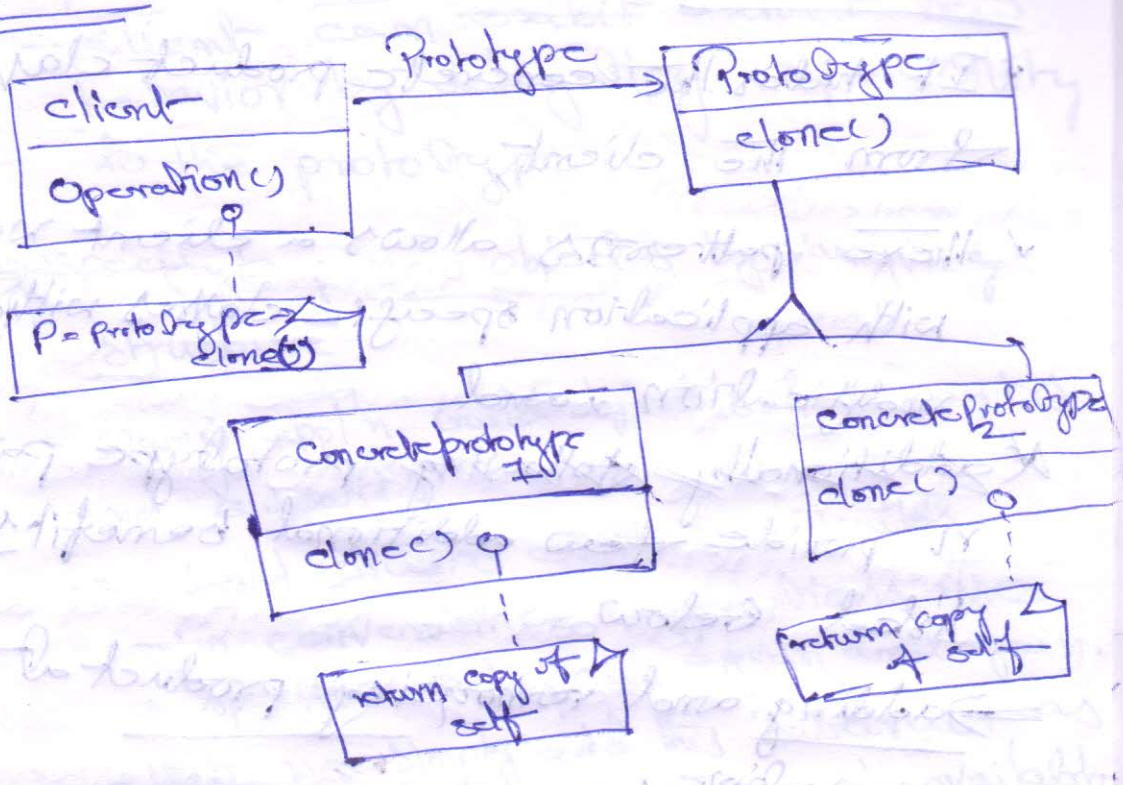
creat

prototype

Collaboration

A clie

Structure



Participants

Prototype –
declares an interface for cloning itself.

Concrete Prototype –
implements an operation for cloning itself.

client
creates a new object by asking a prototype to clone itself.

Collaborations –
A client asks a prototype to clone itself

Consequences

- ✓ It hides the concrete product classes from the client,
- ✓ these patterns allows a client work with application specific classes without modification, and
- ✓ additionally following prototype pattern provide few additional benefits, listed below

⇒ adding and removing product at run-time.

incorporate a new concrete product class into a system simply by registering a prototypical instance with the client.

- ✓ client can install and remove prototypes at run-time.

⇒ specifying new objects by varying values

we can define new behavior through object composition - by specifying values for an object variables for ex: defining new kinds of objects by instantiating existing classes and registering the instances as prototypes

of client objects.
- client can ~~exhibit~~ exhibit new behavior by delegating responsibility to the prototype.

⇒ Specifying new objects by varying structure

- many appl'n build objects from parts and subparts.
ex: build circuits out of subcircuits.
for convenience, applications use a specific subcircuit again and again.
- we simply add this subcircuit as a prototype to the palette of available circuit elements.
- when ever composite circuit object implements clone as a deep copy, circuit with different structures can be prototype.

⇒ Reduced subclassing -

The prototype pattern clone a prototype instead of asking a factory method to make a new object. Hence we don't need a creator class hierarchy at all.

⇒ Configuring an appl'n with classes dynamically :

Implementation

- Using Prototype manager -

clients won't manage prototype themselves but will store and retrieve them from the registry.

- client will ask the registry for a prototype before cloning it.

* prototype manager is an associative store that returns the prototype matching given key.

- Implementing clone operation

- Initializing clones.