CSCI 253

Object Oriented Design:
Creational Patterns
George Blankenship

Overview

Creational Patterns
- Singleton
- Abstract factory
- Factory Method
- Prototype
- Builder

Structural Patterns
- Composite
- Façade
- Proxy
- Flyweight
- Adapter
- Bridge
- Decorator

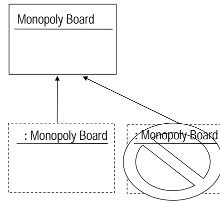
Behavioral Patterns
- Chain of Respons.
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

The Elements of a Design Pattern

- The pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Singleton Pattern: The Problem

Ensure that a class has exactly one instance and provide a global point of access to it



- There can be only one print spooler, one file system, one window manager in a standard application
- There is only one game board in a monopoly game; one maze in a maze-game

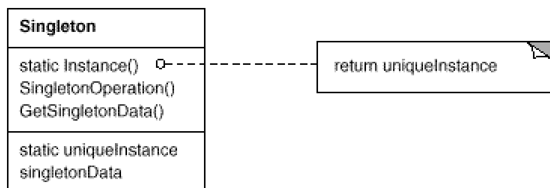
Creational Patterns George Blankenship 4

The Singleton Pattern Participant & Collaboration

- Participant:
- Singleton:
  - is responsible for creating and storing its own unique instance
  - defines an Instance operation that lets clients access its unique instance
- Collaboration:
  - the "class level" Instance operation will either return or create and return the sole instance; a "class level" attribute will contain either a default indicating there is no instance yet or the sole instance

Creational Patterns George Blankenship 5

Control Unique Existance



Creational Patterns George Blankenship 6

Exception Definition

```
class SingletonException extends
   RuntimeException {
    // new exception type for singleton classes
   public SingletonException() {super();}
   // new exception type with description
   public SingletonException(String s) {super(s);}
   }
```

Creational Patterns George Blankenship 7

PrintSpooler Class

```
class PrintSpooler {
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag=false; //true if 1 instance
   public PrintSpooler() throws SingletonException {
        if (instance_flag)
                throw new SingletonException("Only one spooler allowed");
        else
                instance_flag = true; //set flag for 1 instance
        System.out.println("spooler opened");
   }
//-------------------------------------------
   public void finalize() {
        instance_flag = false; //clear if destroyed
   }
}
```

Creational Patterns George Blankenship 8

Print Spooler Creation

```
public class singleSpooler {
   static public void main(String argv[]) {
        PrintSpooler pr1, pr2;
        //open one spooler--this should always work
        System.out.println("Opening one spooler");
        try{pr1 = new PrintSpooler();}
        catch (SingletonException e) {System.out.println(e.getMessage());}
        //try to open another spooler --should fail
        System.out.println("Opening two spoolers");
        try {pr2 = new PrintSpooler();}
        catch (SingletonException e) {System.out.println(e.getMessage());}
   }
}
```

Creational Patterns George Blankenship 9

The Singleton Pattern Consequences

- + Controlled access to sole instance : because the Singleton class encapsulates its sole instance it can have strict control
- + Reduced name space: is an improvement over polluting the names space with global variables that store sole instances
- + Permits refinement of operations and representation: the Singleton class may be subclassed and the application can be configured with an instance of the class you need at run time
- + Permits a variable number of instances: the same approach can be used to control the number of instances that can exist; an operation that grants access to the instance(s) must be provided
- + More flexible than using class operations only

Creational Patterns George Blankenship 10

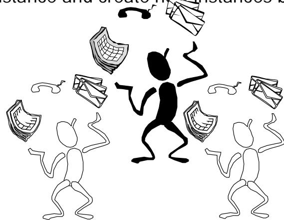
The Singleton Pattern Implementation

- Ensuring a unique instance:
  - the constructors or new operations must be protected or overridden to avoid that other instances are made accidentally by user code
- Subclassing the Singleton class:
  - the main issue is installing a unique instance of the desired subtype at run time
  - when all subclasses are known beforehand the Instance operation can be a conditional and create the right instance depending on some parameter or explicit user input
  - when the subclasses are not known beforehand a register can be used: all subclasses register an instance in it; the Instance operation picks the correct instance out of it

Creational Patterns George Blankenship 11

The Prototype Pattern: The Problem

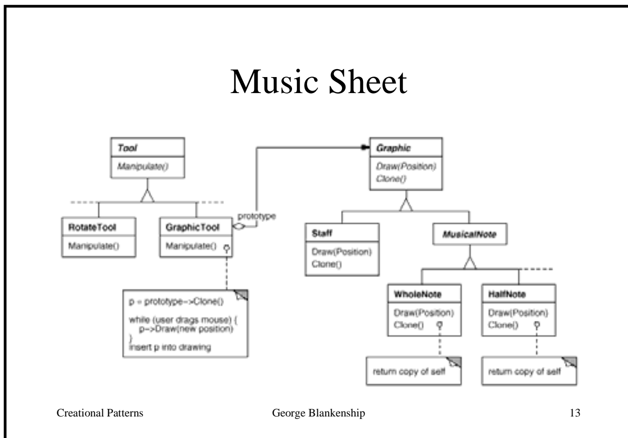
Specify the kinds of objects to create using a prototypical instance and create new instances by copying this prototype

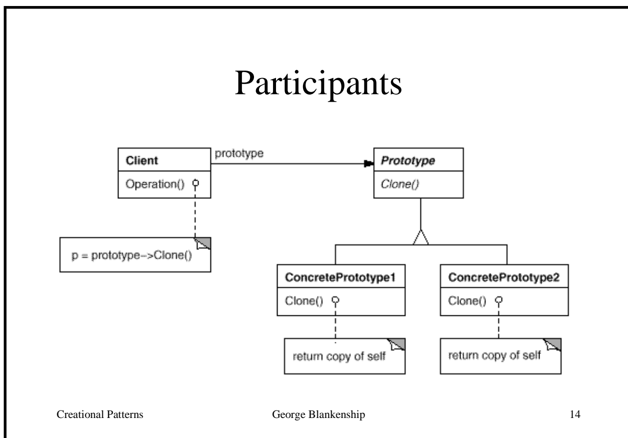


- when an application needs the flexibility to be able to specify the classes to instantiate at run time

- when instance of a class have only very few different combinations of state

Creational Patterns George Blankenship 12





The Prototype Pattern Participants an Collaborations

- *Prototype*: declares an interface for cloning itself
- *ConcretePrototype*: implements an operation for cloning itself
- *Client*: creates a new object by asking the prototype to clone itself

- Client asks a Prototype to clone itself

Creational Patterns George Blankenship 15

Cloning

```
private void sendMessage(String body, HL7Connection connection, int transportId) {
    boolean newMessage = true;
    Message message = new
    Message(mshDataOut.getSentControlID(),connection.getRemoteStation(),body);
    if(message.getMessageACK()) { // does the message require an accept ACK
            Message m = (Message) message.clone();
            connection.queueMessageCx(m); }
     if(message.getApplicationACK()) { // does message require an application ACK
            Message m = (Message) message.clone();
            connection.queueMessageAx(m); }
    if(mshDataOut.getType().equals("ACK") && // is message accept ACK
            ackDataOut.getCode().charAt(0)=='C') {
            newMessage = false;
     } else newMessage = true;
    try {
            connection.sendMessage(message.getControlId(),body);
            if(connection.hasActiveFSM())
            connection.getFSM().execute(FSMeventTypes.FSM_MESSAGE_SENT.ordinal());
     } catch (ChameleonException e) {
            trace.exception(e,"(sendMessage) while trying to send message to remote host");
     }
  }
```

Creational Patterns George Blankenship 16

The Clone

```
//      Clone a copy – variant of constructor for Message
   public Message clone() {
        Message m = new Message(controlId,station,body);
        m.setMessageACK(this.messageACK);
        m.setApplicationACK(this.applicationACK);
        return m;
   }
```

Creational Patterns George Blankenship 17

The Prototype Pattern Consequences (1)

- + Hides the concrete product classes from the client: clients can work with application specific classes without modification
- + Products can be added and removed at run-time: new concrete products can be incorporated by just registering them with the client
- + Specifying new objects by varying values: new kinds of objects are effectively defined by instantiating a specific class, filling in some of the instance variables and registering this as a prototype
- + Specifying new objects by varying structure: complex user-defined structures can be registered as prototypes as well and used over and over again by cloning them

Creational Patterns George Blankenship 18

The Prototype Pattern Consequences (2)

- + Reduced subclassing: as opposed to the Factory Method pattern that often produces a hierarchy of creator classes that mirrors the hierarchy of ConcreteProducts
- + Configuring an application with classes dynamically: when the run-time environment supports dynamic loading of classes the prototype pattern is a key to exploiting these facilities in static languages (the constructors of the dynamically loaded classes cannot be addressed statically, instead the run-time environment creates automatically a prototype instance that the application can use through a prototype manager)
- - Implementing the Clone operation: is difficult when the classes under consideration already exist or when the internals include objects that do not support copying or have circular references

Creational Patterns George Blankenship 19

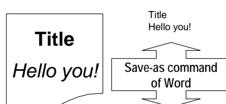
The Prototype Pattern Implementation

- Using a prototype manager: when the number of prototypes in a system is not fixed it is best to use a registry of available prototypes
- Implementing the clone operation: many languages have some support for implementing the clone operator (copy constructors in C++, copy method in Smalltalk, save + load in systems that support these) but in itself they do not solve the shallow / deep copy issue
- Initialising clones: some clients are happy with the clone as it is, others will want to initialise the clone; passing parameters to the clone operation precludes a uniform cloning interface; either use state changing operation that are provided on the clone immediately after cloning or provide a Initialise method
- In languages that treat classes as first class objects the class object itself is like a prototype for creating instances of each class

Creational Patterns George Blankenship 20

The Builder Pattern: The Problem

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



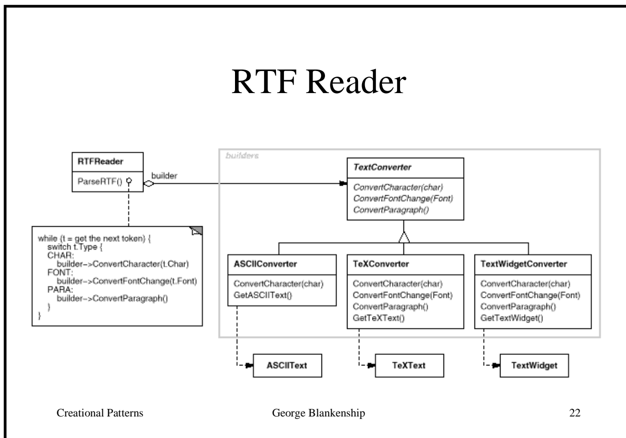
Title
Hello you!

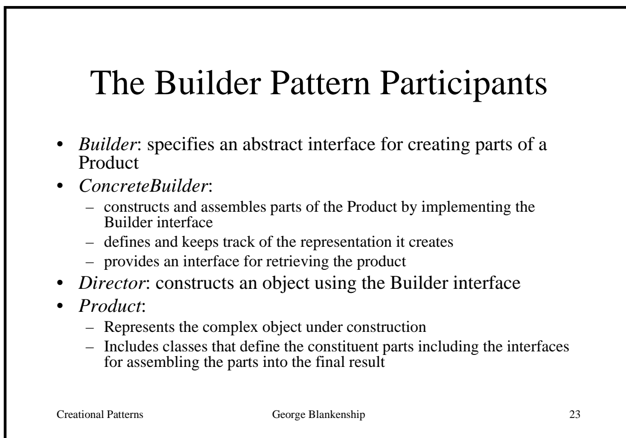
Save-as command of Word

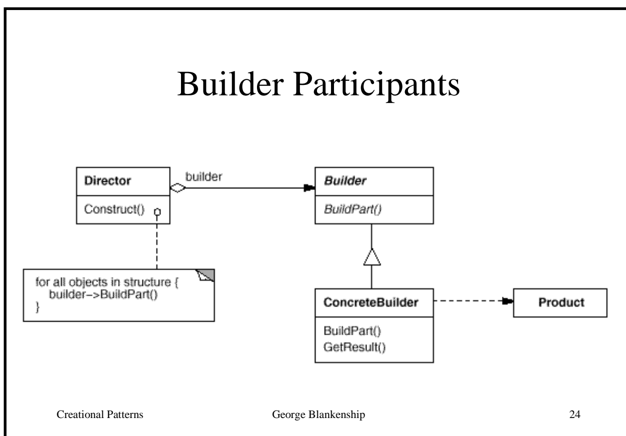
<B><FONT FACE="Arial" SIZE=6><P>Title</P>
</B></FONT><FONT FACE="Times">
</FONT><I><FONT FACE="Arial"><P>Hello you!</P></I></FONT></BODY>
</HTML>

- a RTF reader that can convert into many different formats
- a parser that produces a complex parse tree

Creational Patterns George Blankenship 21


RTF Reader
RTFReader
ParseRTF()
builder
builders
TextConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
while (t = get the next token) {
switch t.Type {
CHAR:
builder–>ConvertCharacter(t.Char)
FONT:
builder–>ConvertFontChange(t.Font)
PARA:
builder–>ConvertParagraph()
}
}
ASCIIConverter
ConvertCharacter(char)
GetASCIIText()
TeXConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()
TextWidgetConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()
ASCIIText
TeXText
TextWidget
Creational Patterns
George Blankenship
22


The Builder Pattern Participants
• Builder: specifies an abstract interface for creating parts of a Product
• ConcreteBuilder:
– constructs and assembles parts of the Product by implementing the Builder interface
– defines and keeps track of the representation it creates
– provides an interface for retrieving the product
• Director: constructs an object using the Builder interface
• Product:
– Represents the complex object under construction
– Includes classes that define the constituent parts including the interfaces for assembling the parts into the final result
Creational Patterns
George Blankenship
23

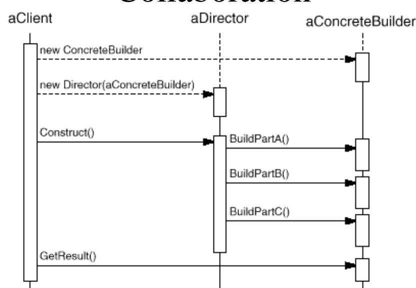

Builder Participants
Director
Construct()
builder
Builder
BuildPart()
for all objects in structure {
builder–>BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product
Creational Patterns
George Blankenship
24

The Builder Pattern Collaboration

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

Creational Patterns George Blankenship 25

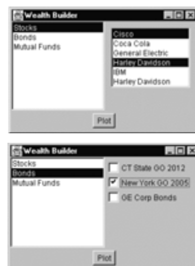
Collaboration



Creational Patterns George Blankenship 26

Multichoice GUI

- We would like to have a display that is easy to use for either a large number of funds (such as stocks) or a small number of funds (such as mutual funds).
- We want some sort of a multiple-choice display so that we can select one or more funds to plot.
- If there is a large number of funds, we'll use a multi-choice list box and if there are 3 or fewer funds, we'll use a set of check boxes.
- We want our Builder class to generate an interface that depends on the number of items to be displayed, and yet have the same methods for returning the results.



Creational Patterns George Blankenship 27

multiChoice Class

```
abstract class multiChoice {
//This is the abstract base class that are the parent for the listbox and checkbox
    choice panels
    Vector choices; //array of labels
//--------------------------------------------
    public multiChoice(Vector choiceList) {
        choices = choiceList; //save list
    }
//to be implemented in derived classes
    abstract public Panel getUI();              //return a Panel of components
    abstract public String[] getSelected(); //get list of items
    abstract public void clearAll();            //clear selections
}
```


Choice Panel Classes

- class listboxChoice extends multiChoice
  - Create a list box for a large number of choices
- class checkBoxChoice extends multiChoice
  - Create a set of check boxes for small number of choices

Panel Generation

```
class choiceFactory {
    multiChoice ui;
//This class returns a Panel containing a set of choices displayed by one of several UI
    methods.
    public multiChoice getChoiceUI(Vector choices) {
        if(choices.size() <=3) //return a panel of checkboxes
                ui = new checkBoxChoice(choices);
        else                   //return a multi-select list box panel
                ui = new listboxChoice(choices);
        return ui;
    }
}
```

Create Connection (Client)

```
/**
 * Creates a new client connection and adds it to the list of connections
 *  Single object to control the priveately created objects
 *
 *  @return connection
 */
public HL7Connection addClientConnection(String host, int port) throws ChameleonException
  {
   HL7Connection newConnection = new HL7Connection(host,port);
   newConnection.client = true;
   newConnection.open = false;
   newConnection.clientSocket = null;
   newConnection.serverSocket = null;
   newConnection.serverClientSocket = null;
   ClientOpen thread = new ClientOpen(newConnection);
   thread.start();
  return newConnection;
}
```


Create Connection (Server)

```
/**
 * Creates a new server listener and adds it to the list of connections
 *  Single object to control the priveately created objects
 *  Starts the listening thread
 *
 *  @param port is the listening port
 *  @return connection
 */
public HL7Connection addServerConnection(String host, int port) throws ChameleonException
  {
   HL7Connection newConnection = new HL7Connection(host,port);
   newConnection.client = false;
   newConnection.open = false;
   newConnection.clientSocket = null;
   newConnection.serverSocket = null;
   newConnection.serverClientSocket = null;
   ServerListen thread = new ServerListen(newConnection);
   thread.start();
  return newConnection;
}
```


The Builder Pattern Consequences

- + Lets you vary the product’s internal representation: the directors uses the abstract interface provided by the builder for constructing the product; to change the products representation, just make a new type of builder
- + Allows reuse of the ConcreteBuilders: all code for construction and representation is encapsulated; different directors can use the same ConcreteBuilders
- + Gives finer control over the construction process: in other creational patterns, construction is often in one shot; here the product is constructed step by step under the director’s guidance giving fine control over the internal structure of the resulting product

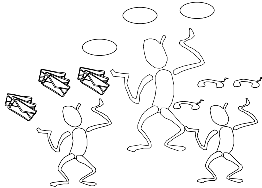
The Builder Pattern Implementation

- Assembly and construction interfaces:
  - The Builder interface must be general enough to allow the construction of products for all kinds of ConcreteBuilders
  - The model for construction and assembly is a key design issue
- Why no abstract class for products?:
  - In the common case, the products can differ so greatly in their representation that little is to gain from giving different products a common parent class
  - Because the client configures the Director with the appropriate ConcreteBuilder, the client knows the resulting products
- Empty methods as default in Builder:
  - In C++ the build methods are intentionally not pure virtual member functions but empty methods instead; this allows clients to overwrite only the operations they are interested in

Creational Patterns George Blankenship 34

The Factory Method Pattern: The Problem

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

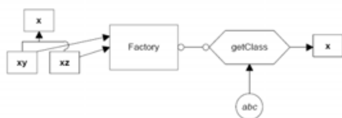


Framework (toolkit) uses abstract classes to define and maintain relationships between objects and is responsible for creating the objects as well

Also known as : Virtual constructor

Creational Patterns George Blankenship 35

Factory Method Mechanics



- ***x*** is a base class and classes ***xy*** and ***xz*** are derived from it.
- The factory is a class that decides which of these subclasses to return depending on the arguments you give it.
- The *getClass* method receives the value *abc*, and that returns an instance of the class ***x***.
- Each instance has the same methods, but different implementations.
- The instance decision is entirely embedded in the factory and could be very complex but is often quite simple; the factory manufactures the object.

Creational Patterns George Blankenship 36

Entry Form

- Entry form that allows the user to enter a name either as “firstname lastname” or “lastname, firstname”
- Simplifying assumption that the name order is indicated by the existence of a comma between the last and first name.

Factory Method Class

```
class NameFactory {
// returns an instance of LastFirst or FirstFirst
// depending on whether a comma is found
    static public Namer getNamer(String entry) {
    int i = entry.indexOf(","); //comma determines name order
    if (i>0)
        return new LastFirst(entry); //return one class
    else
        return new FirstFirst(entry); //or the other
    }
}
```


Factory Method Worker Classes

```
class FirstFirst extends Namer { //split first last
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" "); //find sep space
        if (i > 0) {
            first = s.substring(0, i).trim(); //left is first name
            last =s.substring(i+1).trim();  //right is last name
        } else {
            first = “”; // put all in last name
            last = s; // if no space
        }
    }
}
class LastFirst extends Namer { //split last, first
    public LastFirst(String s) {
        int i = s.indexOf(","); //find comma
        if (i > 0) {
            last = s.substring(0, i).trim();    //left is last name
            first = s.substring(i + 1).trim(); //right is first name
        } else {
            last = s; // put all in last name
            first = ""; // if no comma
        }
    }
}
```

Name Divider Example

Callback for "Enter name:"

Namer name =
  Name.getName(EnterName.getText())

name is LastFirst object

FirstName.setText(name.firstName());

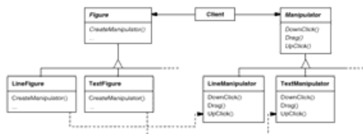
LastName.setText(name.lastName());

Creational Patterns George Blankenship 40

GUI Widgets

Creational Patterns George Blankenship 41

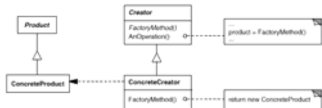
Toolset Generation



Creational Patterns George Blankenship 42

Factory Method Classes

- *Creator* is parent
  - May use static method *FactoryMethod*()
  - May be instantiated as a Factory object
- Set of *ConcreateCreator* classes used to create *ConcreteProduct* (objects)



Creational Patterns George Blankenship 43

The Factory Method Pattern Consequences

- + Eliminates the need to bind application specific classes into your code
- - Clients might have to subclass the Creator class just to create a particular ConcreteProduct object
- + Provides hooks for subclasses: the factory method gives subclasses a hook for providing an extended version of an object
- + Connects parallel class hierarchies: a clients can use factory methods to create a parallel class hierarchy (parallel class hierarchies appear when objects delegate part of their responsibilities to another class)

Creational Patterns George Blankenship 44

Factory Method Pattern Implementation

- Two major varieties are
  - (1) the Creator class is an abstract class and does not provide an implementation for the factory method it declares; the subclasses are required to provide the implementation
  - (2) the Creator class is a concrete class and provides a default for the implementation of the factory method; the factory method just brings the flexibility for subclasses to create different objects
- Factory Methods can be parameterised with something that identifies the object to create (the body is then a conditional); overriding a parameterised factory method makes it easy to selectively extend or change the products that are created
- Use naming conventions that make clear that you are using factory methods

Creational Patterns George Blankenship 45

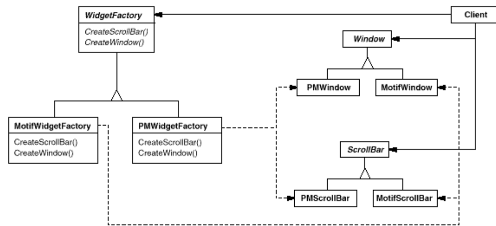
The Abstract Factory Pattern: The Problem

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

- A GUI toolkit that supports multiple look-and-feel standards
- Achieve portability of an application across different windowing systems

Also known as : Kit

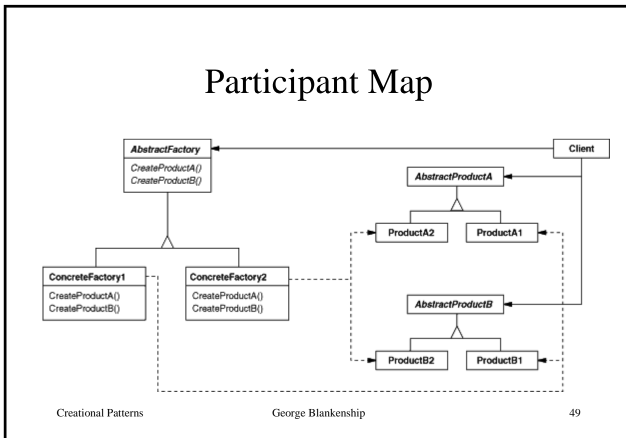
Widget Factory



The Abstract Factory Pattern Participants

- *AbstractFactory*: declares an interface for operations that create abstract product objects
- *ConcreteFactory*: implements the operations to create concrete product objects
- *AbstractProduct*: declares an interface for a type of product object
- *ConcreteProduct*: defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface
- *Client*: uses only interfaces declared by AbstractProduct and AbstractFactory

Participant Map



Creational Patterns George Blankenship 49

The Abstract Factory Pattern Collaboration

- AbstractFactory defers creation of product objects to its ConcreteFactory subclass
- A single instance of a ConcreteFactory is created at run-time; this concrete factory creates product objects having a particular implementation

Creational Patterns George Blankenship 50

UI Look and Feel

```
String laf =
   UIManager.getSystemLookAndFeelClassName();
   try {UIManager.setLookAndFeel(laf);}
   catch (UnsupportedLookAndFeelException exc)
        {System.err.println("UnsupportedL&F: " + laf);}
   catch (Exception exc)
        {System.err.println("Error loading " + laf);}
```

Creational Patterns George Blankenship 51

The Abstract Factory Pattern Conseq. (1)

- + Isolates concrete classes: the AbstractFactory encapsulates the responsibility and the process to create product objects, it isolates clients from implementation classes; clients manipulate instances through their abstract interfaces, the product class names do not appear in the client code
- + Makes exchanging product families easy: the ConcreteFactory class appears only once in an application -that is, where it is instantiated- so it is easy to replace; because the abstract factory creates an entire family of products the whole product family changes at once

Creational Patterns George Blankenship 52

The Abstract Factory Pattern Conseq. (2)

- + Promotes consistency between products: when products in a family are designed to work together it is important for an application to use objects from one family only; the abstract factory pattern makes this easy to enforce
- +- Supporting new types of products is difficult: extending abstract factories to produce new kinds of products is not easy because the set of Products that can be created is fixed in the AbstractFactory interface; supporting new kinds of products requires extending the factory interface which involves changing the AbstractFactory class and all its subclasses

Creational Patterns George Blankenship 53

The Abstract Factory Pattern Implement. (1)

- Factories as singletons: an application needs only one instance of a ConcreteFactory per product family, so it is best to implement this as a singleton
- Creating the products:
  - AbstractFactory only declares an interface for creating products, it is up to the ConcreteFactory subclasses to actually create products
  - The most common way to do this is use a factory-method for each product; each concrete factory specifies its products by overriding each factory-method; it is simple but requires a new concrete factory for each product family even if they differ only slightly
  - An alternative is to implement the concrete factories with the prototype pattern; the concrete factory is initialised with a prototypical instance of each product and creates new products by cloning

Creational Patterns George Blankenship 54

The Abstract Factory Pattern Implement. (2)

- Defining extensible factories:
  - a more flexible but less safe design is to provide AbstractFactory with a single “make” function that takes as a parameter (a class identifier, a string) the kind of object to create
  - is easier to realise in a dynamically typed language than in a statically typed language because of the return type of this “make” operation
  - can for example be used in C++ only if all product objects have a common base type or if the product object can be safely coerced into the type the client that requested the object expects; in the former the products returned all have the same abstract interface and the client will not be able to differentiate or make assumptions about the class of the product

Creational Patterns George Blankenship 55

The Design Patterns

- The Factory Pattern is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory.
- The Abstract Factory Pattern is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes.
- The Builder Pattern assembles a number of objects to make a new object, based on the data with which it is presented. Frequently, the choice of which way the objects are assembled is achieved using a Factory.
- The Prototype Pattern copies or clones an existing class rather than creating a new instance when creating new instances is more expensive.
- The Singleton Pattern is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance.

Creational Patterns George Blankenship 56