Design Patterns in Real Life

Author:	Egon Elbre
Version:	0.01

Table of Contents

Creational

Abstract Factory

families of product objects

Overview

This pattern uses one interface to define the factory of methods for constructing interfaced or abstract classes.

Note

TODO

Examples

Platform independant factory

blah:

```
Filesystem <- Mac FS -> new File with line end set to #10
              Win FS -> new File with line end set to #13#10
```

Different GUI implementations

blah:

```
GUIFactory <- Mac factory -> MacButton, MacPanel
              Win factory -> WinButton, WinPanel
              Anim factory -> AnimatedButton, AnimatedPanel
```

Tips

> ***Note***
>
> TODO

Warnings

> ***Note***
>
> TODO

More

> ***Note***
>
> TODO

Builder

how a composite object gets created

Overview

Classes for the pattern.

Builder:
Abstract interface for creating objects (product)

Concrete Builder:
Provides implementation for Builder.

Director:
Director creates the product by using a Concrete Builder.

Product:
The object that will be created.

This helps to split algorithm of creating something to algorithm and the creation.

> ***Note***
>
> TODO Diagram of the pattern.

Examples

Different file writers

You need to write different file formats and have determined way of order of writing the document.

Builder: table writer Concrete Builder: html writer, csv writer Director: write rows, write columns Product: a document

> ***Note***
>
> TODO Example of not using the pattern

> ***Note***
>
> TODO Example of using the pattern.

Creating complex objects

You need to create a complex object using a abstract factory (see Abstract Factory).

Builder: abstract factory Concrete Builder: Win factory, Mac Factory Director: creates a panel with aligned buttons Product: panel with buttons

> ***Note***
>
> TODO Example of not using the pattern

> ***Note***
>
> TODO Example of using the pattern.

Tips

> ***Note***
>
> TODO Useful tips when to use.

Warnings

> ***Note***
>
> TODO Misuses and bad examples.

More

> ***Note***
>
> TODO Additional information resources.

Factory

easier object creation

Overview

> ***Note***
>
> TODO Description and general idea of the pattern.

> ***Note***
>
> TODO Diagram of the pattern.

Examples

Note

TODO Example

Avoid inheritance

Avoiding useless inheritance and code duplication:

```
Factory.createBlueButton() -> a button colored blue
Factory.createRedButton()  -> a button colored red
```

Note

TODO Principle

Note

TODO Example of not using the pattern

Note

TODO Example of using the pattern.

Information containment

Information containment:

```
Database.createSocket() -> socket with appropriate variables taken
                           from database object
```

Note

TODO Principle

Note

TODO Example of not using the pattern

Note

TODO Example of using the pattern.

Lifetime management

Lifetime management of created objects:

```
WindowManager.createWindow() -> Window, adds window to self
WindowManager.MinimizeAll()  -> calls minimize for each window
```

Avoid object duplication

Avoiding duplicate object creation:

```
ResourceManager.createResource("somefile.jpg") -> FileX
ResourceManager.createResource("somefile.jpg") -> FileX
```

Tips

Note

TODO Useful tips when to use.

Warnings

Note

TODO Misuses and bad examples.

More

Note

TODO Additional information resources.

Factory Method

subclass of object that is instantiated

Object creation based on parameters:

```
Factory.create('./file.txt')   -> File
Factory.create('./directory/') -> Directory

ImageReader.create('./test.jpg') -> JPGReader
ImageReader.create('./test.gif') -> GIFReader
```

Prototype

class of object that is instantiated

Reducing inheritance by using prototype objects instead of classes:

```
Creature -> Orc.clone() -> Creature of class Orc
```

Avoding instantiation of "expensive" classes:

```
Camera = Yaw x Roll x Location x Transformation
for each p
    c = Camera.clone()
    c = c x p()
```

Singleton

the sole instance of a class

Abstract Factory that deals with global lifetime management:

```
WindowManager that deals with all window management
WindowManager.getInstance().createWindow()
WindowManager.getInstance().MinmizeAll()
```

Holding a global state:

```
Application-Wide Clipboard
```

Resource accessing classes:

```
async file - filesystem accesses
zip.open()
zip.edit()
zip.add()
```

Interfacing with modules/devices that have a global state:

```
device.open()
device.read()
device.close()
```

> ***Warning***
>
> Use only if creating a new instance would break something!

If object is single does not mean it has to be a singleton. Using a singleton hides that you are using a global variable.

Bad example:

```
singleton Logger class
```

Structural

Adapter

interface to an object

Use only part of an object for your needs.

Increase usability of modules (composit multiple libraries):

```
db = MathAdaptor (uses different math libraries)
```

Avoid binding to vendor API:

```
db = DBAdaptor
```

Bridge

implementation of an object

Abstract away some part of an object implementation:

```
Content
ContentDrawer
  > ContentDrawerAlpha
  > ContentDrawerBeta

Drawer
DrawingAPI
  > GDI
  > PNG
```

Declare abstract class interface for switchable library:

```
DBAdaptor
db = SQLiteAdapter
db = MySQLAdapter
```

Composite

structure and composition of an object

Use abstract object to define the structure.

Decorate

responsibilities of an object without subclassing

Essentially generic inheritance.

Extend a object with additional functionality:

```
Window + Scrollbars
```

Facade

interface to a subsystem

Provide API for your library.

Flyweight

storage costs of objects

Use a simple object for getting the heavy oject:

```
Font size, style -> shared Font object
```

Proxy

how an object is accessed; it's location

Hide what and how an object is actually accessed.

Object Pool

avoid creating "expensive" objects

Reuse already used objects:

```
Sockets pool (avoids creating sockets)
```

Behavioral

Chain of Responsibility

pass request to object that can fulfill it

Build a tree of handling processing:

```
multiple screen elements
window -> no handle -- pass on to --> panel
panel -> handle -> done
```

Command

when and how a request is fulfilled

Multi-level undo:

```
build list of commands
each command knows how to undo itself
```

Actions that can be called from multiple places + shortcuts, images:

```
delphi
```

Macro recording:

```
each command can be recorded/played
```

Task/thread pool:

```
each task is a separate command
threads take task and execute it
```

Networking:

```
remote procedure calls
```

Interpreter

grammar and interpretation of a language

Math expression:

```
math.calculate("5 + 4 + 1")
```

Interpreted programming language, syntax tree.

Iterator

how an aggregate's elements are accessed, traversed

Unicode string algorithms must work with iterators otherwise incorrect or slow implementation.

Iterating over a set of elements:

```
for x in set:
    print x
```

Generics over lists, trees, sets

Mediator

how and which objects interact with each other

Avoid direct dependancy between classes:

instead of

Client -> Folder

use

Client -> ClientFolderMapping -> Folder

Rules of thumb:

```
http://sourcemaking.com/design_patterns/mediator
```

Memento

what private information is stored outside an object, and when

Restore points - save state to recover from exceptions.

Autosave.

Null Object

Avoid null pointer exceptions while dealing with linked objects:

Tree sentinel objects

Deal easily with exceptional states.

Observer

number of objects that depend on another object; how the dependent objects stay up to date

Update when data changes:

```
data.change --> listbox.datachanged
```

Avoid polling data for changes:

data.change --> listbox.datachanged invalidated screen part -> redraw invalidated part

State

states of an object

Different tools in image editor:

```
Abstract tool
  >  CircleTool
  >  PenTool
```

Finite state machine.

Different contexts of doing something.

Strategy

an algorithm

Different ways of doing something:

printing output format distance function on objects

Function pointers.

Template Method

steps of an algorithm

Provide a default way of doing something to decendant classes.

Queue:

```
put
lock, unlock
get
```

Visitor

operations that can be applied to objects without changing their classes

Printing a tree:

```
vistor.traverse(tree)
```