OO in Delphi – Patterns

Contents

What are patterns?

Software design patterns are named recurring concepts.

We all use them when writing software, whether or not we're aware on them.

Concise terms

We're designing a mechanical device.

We could say:

> We need wheels with pegs that interlock with other wheels with pegs, and we need wheels with extra mass that can maintain rotational momentum.

Or, we could say:

> We need cogwheels and flywheels.

Clearly the second statement is simpler, clearer and more concise.

In essence, that is what patters are: terms that convey concepts – some are simple, some are complex.

Building blocks

When combining concepts, the result will be an even more complex concept.

But if each concept is distinct and named, then combining those concepts becomes surprisingly easy, not just in words but in the mind as well. It's like having conceptual building blocks.

By thinking of patterns as distinct concepts, and combining them, we can quickly come up with more complex architectures with much less re-inventing of wheels.

Example

Consider the following:

"It's a singleton factory that produces TPerson multitons with subscribers."

At the end of this presentation, I will repeat that, and then it will hopefully make sense.

Disclaimer: This is *serious overkill*, but this is a presentation and I'm trying to make a point.

Criticisms

1. Other language paradigms (functional languages, aspect-oriented…) don't need many patterns

Different tools – different strengths.

Other language paradigms have their own patterns (idioms). Patterns still exist, just different patterns.

2. Patterns are used because libraries are incomplete.

Some patterns *can* be written into libraries and simply re-used using inheritance and the like. Some become part of the language. We use them all the time.

Examples:

- containers (TList<>, TDictionary<>)
- abstractions (interfaces, base classes)
- composites (forms/frames with controls and other components)

3. Patterns are too rigid

Understand the *concept* of a pattern. That's what's important about a pattern.

Don't get hung up on "the correct implementation". Some of those implementations are language specific, and none of those languages are Delphi. (C++, Java, C#...)

I can't think of any pattern that has one single right way to be implemented.

Often, pattern implementations are specific to the situation at hand and need to be coded accordingly.

But that's the point; patterns exist precisely because they are *recurring concepts*.

Example:

Going back to the analogy of cogwheels and flywheels…

We can't simply use a standard set of cogwheels because the diameters and cog sizes need to be specific to achieve the right turning rates and engine strength. So we can't rely on a fixed library of cogwheels. We are likely to need cogwheels designed specifically for this mechanical device.

4. People use complex patterns unnecessarily.

People often misuse recently discovered goodies. That's a problem with people, not goodies.

So, when should you use patterns? **When appropriate!**

Pattern are useful tools. Simply using patterns does **not** make one good architect.

A good architect makes good designs, and a good range of patterns makes up a handy toolset.

Disclaimer: This is a presentation on patterns, so I might have gone overboard… just a tad.

Creational patterns

Singleton

This is probably the simplest of all patterns. It's almost too simple. All it means is that there will never be more than *one single* instance of something.

Example: global variable

```
type
  TGlobal = class
  public
    UserID  : integer;
    UserName: string;
  end;

var
  Global: TGlobal;
```

Example: global function

```
type
  TGlobal = class
  public
    UserID  : integer;
    UserName: string;
  end;

function Global: TGlobal;

implementation

var
  _Global: TGlobal = nil;

function Global: TGlobal;
begin
  if not Assigned(_Global) then
    _Global := TGlobal.Create;
  Result := _Global;
end;
```

Example: static class

```
type
  Global = class
  public
    class var UserID  : integer;
    class var UserName: string;
  end;
end.
```

Factory

If a shoe factory creates shoes and a car factory creates cars, then an object factory creates objects.

A factory is simply something that creates objects. A simple factory will be quite specific in that it creates a certain type of object.

Example: factory object

```
type
  TPersonFactory = class
  public
    function  NewPerson: TPerson;
    function  GetPerson(const PersonID: TPersonID): TPerson;
  end;

var
  PersonFactory: TPersonFactory;
```

PersonFactory can either produce brand new TPerson objects (using NewPerson() method) or produce specific person objects by supplying a PersonID (using GetPerson() method).

Abstract Factory

Say you want a factory that can create different types of things. Perhaps they're different classes that share the same ancestor. Perhaps they're different implementations of an interface. Whatever they are, one factory simply won't do, because they are different things.

But you still want one central place to obtain those different kinds of objects – one "factory".

An **abstract factory** does not itself create things. Instead, it contains a collection of **concrete factories** (the adjective "concrete", meaning actual/proper) and it then calls the appropriate concrete factory to do the actual creation.

Example: different types of employment

Employees might be employed in different ways: full time, contractor, casual, etc. The employment types are all so different that one class will not fit all. We need a different class for each employment type.

We don't want one single factory to create and load and save all the different variations of employments, as that would make for some complex and messy code. We are better off having several concrete factories, where each one specialises in one employment type. That will keep each concrete factory as simple as possible.

But, to keep the rest of our code simple, we still would prefer a single factory-like object. So, we create an object that looks and behaves like a factory, but is actually a container of concrete employment factories. This is our abstract Employments factory.

Register

This is a pattern I've defined. It would probably be classified as a creational pattern.

It's similar to a dictionary in that you add things to a **register**.

It is *not an abstract factory* because it does not produce new objects.

It is *not a dictionary* because with a dictionary you associate each thing with a simple key. Registers usually use more complex methods of selection.

In some cases, registers might do their own examination of the selection parameters to work out a key to be used on an internal dictionary. In this case the register acts as an Adapter for the dictionary.

In other cases, registers might use a Chain of Responsibility pattern to ask the registered objects themselves to determine which of them to retrieve/use.

In fact, an Abstract Factory could be considered a Register of Concrete Factories.

Multiton

With a **singleton** class, we have only **one single instance** of that singleton class.

With a **multiton** class, we can have **multiple unique instances**. Each instance uniquely represents its own "thing".

Generally, instances are uniquely identified using some kind of ID, such as a number or a GUID.

Example: multiton

```
type
  TPerson = class
  public
    constructor Create(const APersonID: TPersonID);
    property  PersonID      : TPersonID   read fPersonID;
    ...
  end;
```

We can have many instances of TPerson in memory at any given time.

If multiple TPerson objects *represent the same person* then TPerson is *not* a multiton.

If all TPerson objects are *unique* (ie. we don't have multiple TPerson objects that represent the same person) then TPerson is a multiton.

Multiton Factory

What's a good way to ensure that we have multitons? Well, we need some way to prevent developers, including ourselves, from creating duplicates.

The best way is to use a factory that gives us the objects that we want while ensuring that the objects are unique.

However, because a multiton factory might return an already existing object instead of creating a new one, I like to name such methods "Obtain".

Also, because a multiton factory might not actually free a multiton object immediately, I tend to name such methods "Release".

Reference counting multitons

Whenever some code has finished using an object, then it ought to release the object. The point of a multiton object is that it might be used independently in several places in the program at once. When one place has finished using the multiton object then it should release the use of that object, but if the object is still in use elsewhere then we can't actually free the object yet.

So, we need to keep track of how many times each multiton is obtained. The most common way is for the multiton to have its own reference counter. Because the multiton is managed using a multiton factory, the factory can be responsible for changing and checking the reference counter.

Structural and behavioural patterns

Collection

A collection is an object that contains a bunch of things, such as a list, array, queue or stack.

Example:

```
type
  TPerson = class
  private
    ...
  end;

  TPeople = class(TObjectList<TPerson>);
```

An object of type TPeople is a collection that holds TPerson objects.

Composite

Sometimes you have an object that contains other objects, not as a collection, but in a more way such as a tree structure. This is called a **composite**.

Delphi forms and frames could be considered examples of this; each frame/form can be seen as a single thing that consists of a tree of components. There are better examples that I'll show in the demos.

Iterator

An **iterator** steps through the items in a collection.

In Delphi, the generics TList<>, TStack<>, TQueue<>, etc. provide implementations of TEnumerator<T>. An instance of TEnumerator<T> is an **iterator** obtained using a special method called GetEnumerator.

This is what enables:

```
  for var Element in Collection do
```

Delphi finds a method…

```
function GetEnumerator: TEnumerator;
```

…in Collection, calls it to obtain the iterator (of type TEnumerator or a bound TEnumerator<T>), and then the for-loop uses the iterator to traverse the items in the collection.

Chain of responsibility

This a behavioural pattern that is usually used on collections.

The idea is to iterate through the collection until you find the element you're looking for. That's the element that will do (or has done) the job.

Flyweight

You're in a situation where various parts of your software need to use the same data. But why use up memory for each when they can share?

Enter the **flyweight**. A flyweight is an object that holds information that can be used by various parts of an application.

The most common example is probably TImageList. You load up a set of images into an image list, once. Those images can then be shared by buttons, tab items, etc. That way we don't waste time and memory loading the images repeatedly.

Strategy

Imagine a program that may have a connection to a server… or not. We won't know until we run it. Now, if we do have a connection, then we can ask the server for up-to-date settings. But if we don't have a connection, then we could load the settings from file instead. Hmm… which **strategy** to use?

A way to do this is to have an interface as a global variable. There will be different implementations of that interface. At runtime we choose which one of those implementations to use and assign it to the global variable.

Other parts of the program use the global variable, not knowing nor caring which strategy was chosen.

Decorator

Say you have objects that can have various behaviours. You could hard code those behaviours into their classes. You could elect which behaviours should be active using states. Perhaps subclasses or sibling classes could have different behaviours. This can get very messy.

But what if you could distil these behaviours into little objects that you could optionally add on to your objects.

The main object would be a "plain" object to which you add **decorators** (those little objects with distilled behaviour) thereby making your main object more interesting.

The decorators add that extra functionality without the need to alter the original "plain" class.

Dependency injection

There are three parts:

1. **Client** Does some work.
2. **Service** Contains information and/or functionality that the client uses.
3. **Injector** Code that chooses a service and calls the client, passing the service as a parameter.

You could think of the service as a kind of local Strategy that is passed into the client.

Observer/Subscriber

(This pattern has two names.)

Think about a typical "Delphi event" such as a button click or a form create event. The OnClick event on the button and the FormCreate event on the form are both method pointers that typically point to methods within the form.

Because the events are simple method pointers, each one can only point to only one method.

Now imagine being able to point to multiple methods. That is what the Observer pattern enables.

Structure

Say there's an event that is of interest to other parts of the program. Let's call it the **subject**, as it's the subject of interest or attention. Perhaps the event fires upon a state change or some incoming data.

All those parts of the program that want to be notified when the event fires can **subscribe** to the event.

While subscribed, they are called **observers**.

Memento

This is classic Undo/Redo functionality.

This works by making a backup (or multiple) of the state of… something. I say "something" because it could be the values in a single object, or the state of a collection or composite of objects.

If or when you decide to return the "something" to an earlier state, you reset it by making it match the state of an earlier backup.

More…

Of course, there are more patterns. Perhaps you have your own patterns that aren't official, such as my Register pattern. There is a whole category of concurrency patterns that I didn't get to.

Some googling will help you find the most common patterns, but be aware that not all patterns will be useful or suitable for use in Delphi.

Remember that patterns are simply recurring concepts with names.