GDK software

DELPHI PROGRAMMING CHECKLIST

Start writing better software

Author:
Marco Geuze- Embarcadero MVP

MEET MARCO

Marco Geuze is the co-owner of GDK Software and an Embarcadero Delphi MVP. He started developing in the DOS era, writing applications with Turbo Pascal. Since then, he has always written code, moving from Object Pascal to Delphi. He regularly publishes blogs, gives webinars, and has a mailing list to which many Delphi developers are subscribed.

His company GDK Software is an authority on the programming language Delphi (and Mendix, but that is out of scope of this guide) and has helped hundreds of companies with Delphi development, maintenance, and Delphi upgrades. Check out www.gdksoftware.com/knowledgebase for various Delphi related content, and head to www.gdksoftware.com/delphi-hero to subscribe to the GDK Delphi mailing list.

WHAT'S IN THIS CHECKLIST

INTRODUCTION

Delphi (or rather Pascal) is an easy language to learn. There are good reasons why many schools used to teach Pascal to students. The clear structure, simple syntax and speed of development make it a popular language for both students and teachers.

I also came into contact with Pascal in this way, starting with Turbo Pascal. This implementation of Pascal was marketed by Borland and was at that time very successful, mainly because Borland Pascal was very cheap. Here in the Netherlands, you could buy the development environment for just 100 Guilders (about 50 euros).

The great thing about Pascal was that it is very easy to get started. Especially when Delphi was widely used, and RAD (Rapid Application Development) became popular, many applications were developed. Unfortunately, there is a downside to this speed: it is also easy to write code in such a way that it becomes a mess of intertwined functions, procedures, and dependencies.

With this Delphi programming checklist, I hope to give you some tools to write code that is faster, better and above all, more structured. The topics range widely from tools and plugins to technical methods and programming techniques. But they are all aimed at making your life easier. Enjoy!

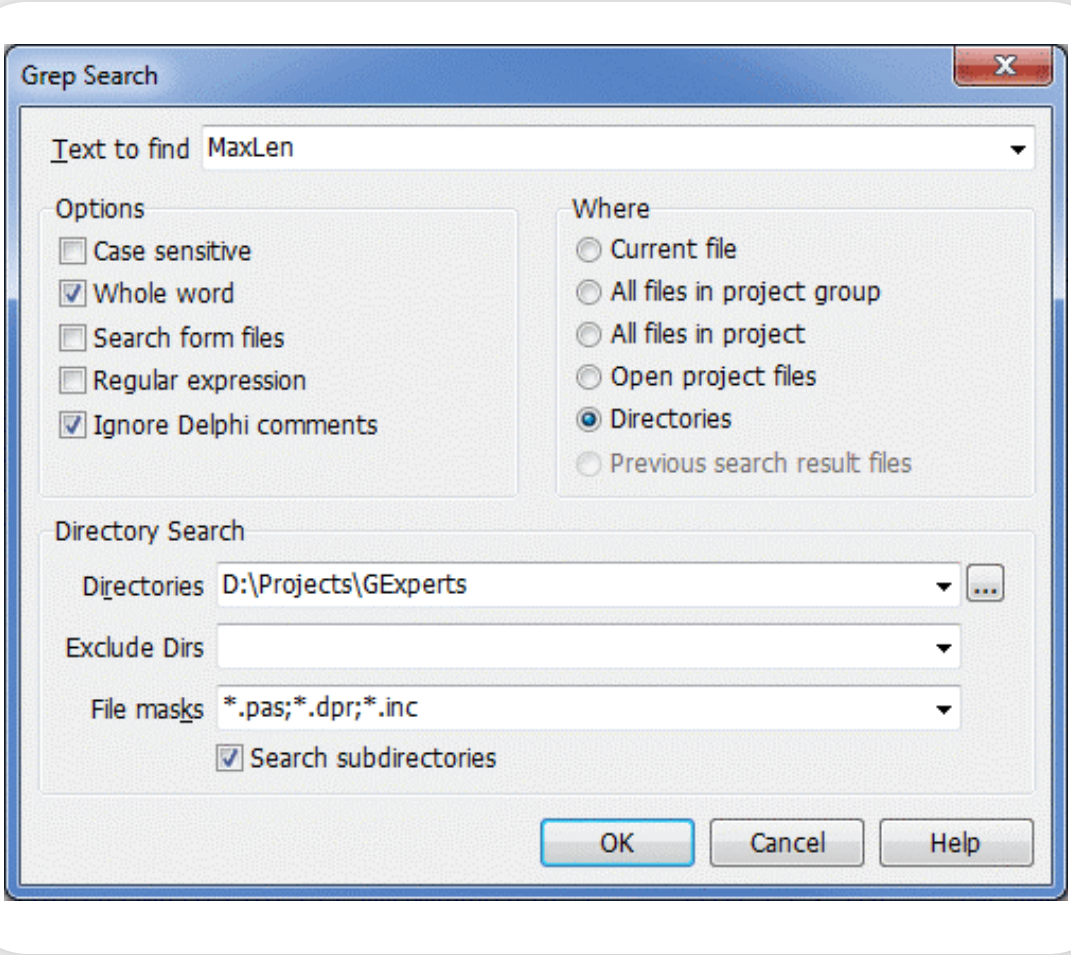
Marco, 2022

USING THE RIGHT TOOLS

Delphi is, out-of-the-box, a great tool for lightning-fast development. However, if you have large projects, it may make sense to install some external plugins. Not only does this allow you to navigate through your code faster, but it can also help you structure your applications. At GDK Software, we usually use the following plugins:

GEXPERTS

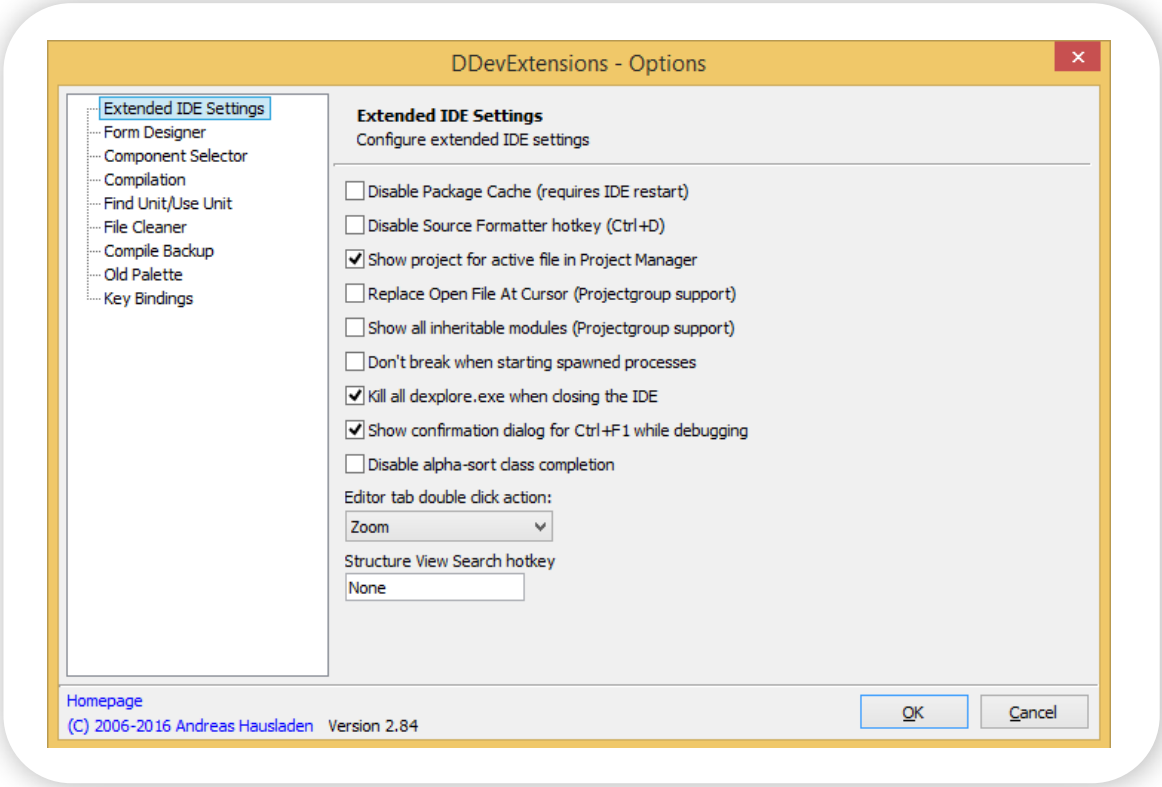
GExperts adds some very useful features to Delphi, such as **GrepSearch** Alt + Shift + S, **Procedure List** Ctrl + G, **RenameComponents** and other very handy shortcuts. We will get to these shortcuts in Chapter Two. Take a look at the tools **Components to Code**, and **Procedure List** as well.



Download: https://link.gdk.software/gexperts

DDEVEXTENSIONS

We love DDevExtensions for the Find Unit/Use Unit replacement dialog (Alt + F11). The default Delphi Use-Unit dialog only shows the files from the project group. DDevExtensions replaces the dialog with one that not only shows the project group files but also all files that the compiler can see. Never again navigate to the uses section and insert a unit by hand – just hit Alt-F11.



Download: https://link.gdk.software/ddevxtensions

SPRING FRAMEWORK

The Spring framework is an open-source code library with a very useful set of interfaced based collection classes, for example, the generic interfaced IList, dictionaries, and so on. It also contains a great dependency injection framework and various encryption libraries.

Download: https://link.gdk.software/spring

MADEXCEPT

This tool is used for sending information and data about crashes and exceptions that happen in the production environment to the developers. Another tool that can be used for this is Eurekalog. I used both and I don't have a clear preference. But at least: use one of them, it will help you find the source of bugs.

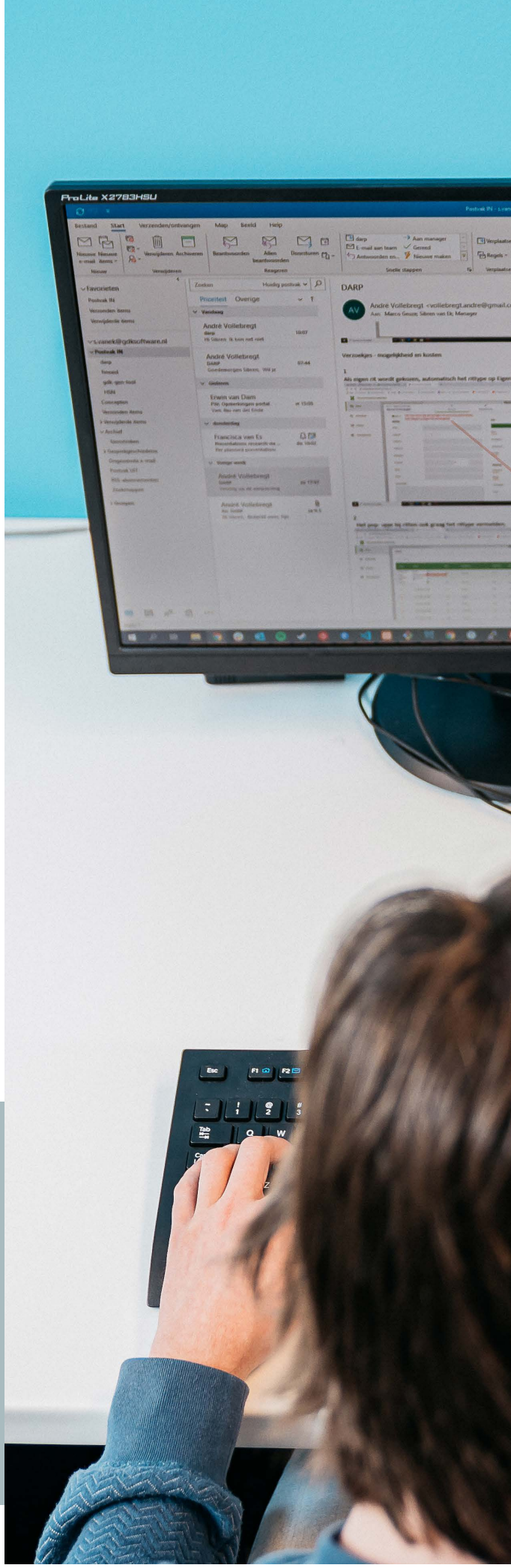
Download:
https://link.gdk.software/madexcept
https://link.gdk.software/eurekalog

OTHERS WORTH MENTIONING

MMX: https://link.gdk.software/mmx

CN Pack: https://link.gdk.software/cnpack

Testinsight: https://link.gdk.software/testinsight



- [ ] Are you using the right tools for the job?
- [ ] Are you up-to-date with Delphi? Embarcadero invests a lot in quality improvements and bug fixes, and the latest Delphi environment is a lot better than Delphi 5 or Delphi 7. Use the guides provided by Embarcadero, or seek help from our Delphi experts via **https://gdksoftware.com/services/delphi-upgrades-and-updates.**

STAY AWAY FROM THE MOUSE!

IDE Shortcuts are the easiest way to speed up development. Undoubtedly, most of us know the most important shortcuts in Delphi, but let's share which ones I use regularly.

Bookmarks

Save the points in code where you want to comeback. Navigate easily through code. Set a bookmark with Ctrl +Shift + <Number> and use it with Ctrl + <Number>.

Jump between the methods interface and implementation with Ctrl + Shift + Arrow up or Ctrl + Shift + Arrow down. Install and use the **Grep Search** from GExperts, searching faster and representing the results in a better view than the Delphi IDEvia Alt + Shift + S. (GExperts) **Search and go** to a method with Ctrl + G. Use Alt + F11 to add a unit to your code (use with DDevExtensions).

Use a clipboard manager to copy several things without continuously swapping between copy/paste, copy/paste, etc. We prefer **CLCL** (https://link.gdk.software/clcl) for clipboard history.You can easily use it by pressing Alt + C. Despite this not being a specific Delphi suggestion, I think it will be of great help.

OTHER POPULAR SHORTCUTS ARE

The obvious ones

- Save: Ctrl + S and save all: Ctrl + Shift + S
- Run: F9, compile: Ctrl + F9 and build: Shift + F9
- Stop run / debug: Ctrl + F2
- Add a file to the project: Shift + F11
- Complete a class: Ctrl + Shift + C (i.e., create the methods in the implementation section).
- Declare a variable: Ctrl + Shift + V
- Declare a class-variable: Ctrl + Shift + D (same as V, but instead of the local variable you'll end up with a class-variable).
- Rename a variable: Ctrl + Shift + E
- Go to unit: Ctrl + Enter (when your cursor is on a unit name). Faster than Ctrl + click, but only works if Delphi can locate the unit.
- Align your codelines with Ctrl + Alt + Z (GExperts).
- Synchronize procedure or function parameters: Ctrl + Alt +Shift + P

- [ ] Are you using the keyboard as much as possible??
- [ ] Check the full list of shortcuts: https://link.gdk.software/shortcuts

CLEAN CODE

It's impossible to overstate the importance of Clean Code. Writing clean code makes your code more readable, more maintainable and easier to understand. If you haven't read the book 'Clean Code' by Robert C. Martin, I would encourage you to do so. It really helps you to write better code.

The most popular definition of clean code is code that is easy to understand and easy to change. Although this might seem simple, it's quite hard to execute. To keep your code simple, you should reduce complexity as much as possible. This means that simpler is always better. However, you should never sacrifice readability or maintainability in order to make your code look simpler.

There are some general lessons to learn if you want to write clean code. As a Delphi developer you probably work with both new and legacy code. Specifically for legacy code, the Boy Scout rule applies: Leave the campground cleaner than you found it. Make sure to improve code when you are refactoring. Split massive functions into smaller ones; give variables sensible names; and make the code readable. After all, you spend a lot of your time reading code, not writing! So why not help yourself and others to make the reading part easier?

> **Truth can only be found in one place: the code**
> *– Robert C.Martin*

To keep your procedures and functions clean, try to keep them as small as possible, and let them do just one thing. If your function is doing more than "one thing," it is a perfect moment to extract that to another function. That's one of the reasons we do not mix business rules with GUI code. We do not mix SQL queries with communication protocols. We keep code that is changed for different reasons separate so that changes to one part do not break other parts. In Delphi this means that you don't write your business logic behind the on-click of a button, but also that you don't use one datamodule for both retrieving data from a database and generating reports.

Procedures and functions should also have as few parameters as possible. Zero would be best, one or two is okay, three and above, not. Because if a function or procedure requires more than 3 parameters, it's likely that some of these parameters could be wrapped in one class.

Use easily-pronounceable names for variables and functions and procedure, and don't use abbreviations. Maybe you save a second to type in AddCmt function, but it looks horrible, and it's not easy to remember or to search for.

```
procedure AddCmt(const M: string);
                    ↓
procedure AddComment(const AMessage: string);
```

Also choose a name to show your intention. The purpose of the variable should be understandable to someone reading that name. Write the name as you would speak it.

```
D: TDateTime; // Elapsed time in days
                  ↓
ElapsedTimeInDays: TDateTime;
```

Be consistent. Use one word for similar functions. Don't use "get" in one class and "fetch" in another.

Comments are difficult to maintain and don't tell the truth about the code, so try to avoid them. They are almost always out of date or redundant. With the help of the tips I showed already, your code should explain itself.

Also, don't comment out unused code, just delete it. You have a source control system anyway (you have one, right?). The only reason to ever use comments is to express the importance of certain points in the code.

Single-letter names pose a particular problem in that they are not easy to locate across a lot of text. Single-letter names can only be used as local variables inside very short procedures. The length of a variable name should correspond to the size of its scope.

And lastly, avoid encodings like putting L in front of local variables. At GDK we also don't use the A Prefix with parameters. The only exception to this rule for us is to private Fields of a class, where we do use the F prefix.

box: `iCounter: Integer;` → `Counter: Integer;`

> **The proper use of comments is to compensate for our failure to express ourselves in code. Note that I used the word 'failure'. I meant it. Comments are always failures.**
>
> *– Robert C.Martin*

- [ ] Programming is a craft. Do you treat it like as such?
- [ ] Have you made your functions, procedures and classes small enough?
- [ ] Does your code explain itself?

SOLID PRINCIPLES

In software engineering, SOLID is an acronym for five designprinciples intended to make software designs more understandable,flexible, and maintainable. The SOLID principles are made famous byRobert C. Martin. Although he invented most of the principlesthat hepromotes, the LiskovSubstitution Principle was invented by BarbaraLiskov, while the Open-Closed Principle was invented by BertrandMeyer.

The five principles are:

1. **The Single Responsibility Principle:** Classes should have a single responsibility, and thus only a single reason to change.
2. **The Open-Closed Principle:** Classes and other entities should beopen for extension but closed for modification.
3. **The Liskov Substitution Principle:** Objects should be replaceableby their subtypes.
4. **The Interface Segregation Principle:** Clients should not be forcedto depend upon interfaces that they do not use.
5. **The Dependency Inversion Principle:** Depends on abstractionsrather than concretions.

SINGLE RESPONSIBILITY PRINCIPLE

Let's start with the Single Responsibility Principle. As the name suggests, each class in a program must have a single responsibility for only one part of the program's functionality. But that seems easier than it is. What exactly is one part of a program, and how to know when to separate functionality? It's too simple to say that a class should only do one thing.

Robert C. Martin expresses the principle as: 'Gather together the things that change for the same reasons. Separate those things that change for different reasons', and more recently 'This principle is about people'. That should point us in the right direction.

When you write a software module, you want to make sure that when changes are requested, those changes can only originate from a single person, or a single, tightly-coupled group of people representing a single, narrowly-defined business function. This means that a software module or class should have one responsibility for that particular group of people.

It is easier to explain this by means of an example. Let's look at this following class:

```
type
  TShip = class
  private
    FPosition: TPoint;
    FHeading: Integer;
    FSpeed: Integer;
    FCargoLoad: string;
  public
    procedure SetHeading(NewHeading: Integer);
    procedure SetSpeed(NewSpeed: Integer);
    function GetCoordinate: TPoint;
    procedure PlotCourse;
    procedure LoadCargo(NewCargo: string);
    procedure PrintCargo;
    procedure ReportPosition;
    procedure CalculateProfit;
  end;
```

This is a class that is doing a couple of things, all about managing a ship's position and course, its load, and some reporting things. As this is a brief example to show how to think about the Single Responsibility Principle, don't pay too much attention to the details of the code itself; it is all about the large overview of the structure of this particular class.

I think we all know these kind of 'God' classes – usually packed with lots of functionality and code, and managing one particular part or module of your program. The question is: if we need to make some changes to this class, how can we refactor this class to make sure we gather together the things that change for the same reason, and separate those things that change for different reasons.

Let's stop for a moment and think about the responsibilities of this code regarding the [group of] people. We can define a couple of specific people that will have some responsibility regarding the ship's management, direction and heading, and reporting tools. So, let's say in this case we define a skipper, a navigator, a cargo-load master and a financial manager. If you think of these different roles, it's suddenly very easy to separate this class into different modules, with just one responsibility for each particular role. We should have a class for setting the heading and power (skipper), one for managing position and plotting the course of the ship (navigator), one for managing the load of the ship (cargo-load master), and one for all our financial reporting (financial manager).

Our new classes might now look like this:

```
type
  TShipLocation = class
  private
    FPosition: TPoint;
  public
    function GetCoordinate: TPoint;
    procedure PlotCourse;
    procedure ReportPosition;
  end;

TShipMovement = class
  private
    FHeading: Integer;
```

```
    FSpeed: Integer;
  public
    procedure SetHeading(NewHeading: Integer);
    procedure SetSpeed(NewSpeed: Integer);
  end;

  TCargo = class
  private
    FCargoLoad: string;
  public
    procedure LoadCargo(NewCargo: string);
    procedure PrintCargo;
  end;

  TShipReport = class
  public
    procedure CalculateProfit;
  end;

TShip = class
  private
  public
    // reference to subclasses
  end;
```

Would you have done the same if you didn't think of the people behind the class's responsibilities? Maybe, but I can imagine the ShipLocation and ShipMovement class could have ended up in the same class.

So, what happens now if we got a feature request from the skipper to add a bow thruster to make it easier to steer the ship in small canals? Just make this change in the ShipMovement class, without affecting any of the other classes. And if we want to implement a new cargo-load system? Just alter the Cargo class, again without touching any of the other classes.

I hope by now you see why the Single Responsibility Principle is really about people, or actors, and the responsibility of the functionality of modules or classes of your program in relation to these people. And of course, you can apply this on different levels of your program, from modules to classes to specific functions.

OPEN-CLOSED PRINCIPLE

The second principle of SOLID is the Open-Closed Principle: 'Classes and other entities should be open for extension but closed for modification'. This article is about how in Delphi you keep a class closed for modification, but open for extension.

Let me emphasise one part; achieving this goal via inheritance or overriding classes is, in my opinion, a bad idea. If you want to read more about this, check out what Bertrand Mayer and Robert C. Martin have to say on this subject.

Okay, let's take the following example:

```
type
  TShip = class
    // This is the actual ship
  end;

  TPartList = class
    // This class holds all the needed parts for building a ship
  end;

  TShipLayout = class
    // This class provides the blueprint of the ship to build
    function GetBlueprint: TList<TPoint>; // just as an example
  end;

  // This is the factory class to build a ship
  TShipBuilder = class
  private
    FParts: TPartList;
    FShipLayout: TShipLayout;
  public
    procedure LoadParts(APartList: TPartList);
    procedure LoadShipLayout(ALayout: TShipLayout);

    function BuildShip: TShip;
  end;
```

As you can see, these are simple classes without interfaces and without the possibility to extend them easily. The ShipBuilder class is, however, already [partly] closed for modification; the internal operation is separated from the outside world because the local variables are made private. In Delphi, however, you can still access these variables when you access this class from the same file. Therefore, a simple improvement is to make the 'private' a 'strictly private'.

However, the inner workings of this class itself is still open for modification. One can easily call on the BuildShip function before even providing the necessary parts and ship layout.

But how do we ensure that these classes are open for extension, but remain closed for modification?

The answer? Interfaces!

Why interfaces? This has to do with another principle: program against interfaces and not against implementations. The example above is the implementation of the class TShipBuilder. As soon as you start working with this implementation, you are automatically 'stuck' with the TPartList and TShipLayout implementations as well. Let's see how we can improve this. The first step is to create several interfaces and modify the TShipBuilder class to implement those new interfaces. The second step is to use dependency injection to open this class for extension but close it for modification. If you do that, you will end up with something like this:

```
type
  TShip = class
    // This is the actual ship class
  end;

  TPartList = interface
    // Necessary info to provide the parts
  end;
```

```
  TPartList = class(TInterfacedObject, IPartList)
    // This class holds all the needed parts for building a ship
  end;

  IShipLayout = interface
    // Necessary info to provide the blueprint
    function GetBlueprint: TList<TPoint>;
  end;

  TShipLayout = class(TInterfacedObject, IShipLayout)
    // This class implements the blueprint function of the ship
to build
    function GetBlueprint: TList<TPoint>;
  end;

  TShipBuilder = class
  strict private
    FParts: IPartList;
    FShipLayout: IShipLayout;
    procedure LoadParts;
    procedure LoadShipLayout;
  public
    constructor Create(APartList: IPartList; ALayout:
IShipLayout);
    function BuildShip: TShip;
  end;
```

So, what exactly did we do? As you can see, we now have a TShipBuilder class which requires the two interfaces via a constructor. Both the LoadParts and LoadShipLayout procedures are made strict private, which ensures that the class is closed for modification, so we use these procedures only within the BuildShip function for example.

Additionally, our class is open for extension. It is possible to extend this ShipLayout functionality, for example, by just creating another implementation of the IShipLayout interface, as long as we implement the GetBlueprint function. We just have to make sure that we will provide a class with the IShipLayout interface implemented when calling the constructor of the TShipBuilder class.

As with most examples I've shown earlier, be aware that you shouldn't put both the interfaces and classes all in the same file. It's just for the sake of readability that these are in one place right now.

So, to summarise: to use the Open-Closed Principle in Delphi you need interfaces to make a class open for extensions and make use of the (strict) private keyword to keep the class from modifications. Use as little inheritance as possible, because with inheritance, you risk opening the class for modifications. And remember: always program against interfaces instead of implementations.

THE LISKOV SUBSTITUTION PRINCIPLE

The next principle is The Liskov Substitution Principle in Delphi! I start with the official definition:

Subtype Requirement: Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.

Still here and wondering what that means? Good, I had some trouble understanding this too.😊

Let us take a more practical approach to LSP: The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. Or, to stay in Delphi terms: If TChild is of a subtype of TParent, then objects of type TParent may be replaced with objects of type TChild, without breaking the logic of the program.

This means that the underlying classes must work in approximately the same way as the parent class. In contrast to the previous SOLID principles we have dealt with, this one is more about the behaviour of classes, and not directly about the structure of these classes. Let's get started again with an example.

```
TEmployee = class
  strict private
    FName: string;
    FManager: TEmployee;
    FSalary: Double;

    procedure SetName(const Value: string);
    procedure SetSalary(const Value: Double);
    function GetName: string;
    function GetSalary: Double;
  public
    procedure AssignManager(const AEmployee: TEmployee);
virtual;

    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;
  end;
```

As you can see, this is a very simple Employee class, which we, for example, could use like this:

```
var
  Employee: TEmployee;
  Manager: TEmployee;
begin
  Manager :=  TEmployee.Create;
  Manager.Name :=  'Jane Smith';
  Manager.Salary :=  31000;

  Employee :=  TEmployee.Create;
  Employee.Name :=  'John Smith';
  Employee.Salary :=  22500;
  Employee.AssignManager(Manager);
end;
```

So far, so good. But let's say we want to add some functionality to the Manager, as he has to do an appraisal for an employee. Maybe you remember that I've mentioned the preference to program against interfaces, not implementation. But for the sake of this example, we're going to override our TEmployee class to see what the Liskov Substitution principle is, before we refactor this again to interfaces. So, let's create the following class:

```
TManager = class(TEmployee)
public
    procedure DoAppraisal(const AEmployee: TEmployee);
  end;
```

This is just fine, as we added some functionality. We can now simply change our implementation to this:

```
var
  Employee: TEmployee;
  Manager: TEmployee;
begin
  Manager :=  TManager.Create;
  Manager.Name :=  'Jane Smith';
  Manager.Salary :=  31000;

  Employee :=  TEmployee.Create;
  Employee.Name :=  'John Smith';
  Employee.AssignManager(Manager);
  Manager.Salary :=  22500;
end;
```

We changed the implementation of the Manager object to a TManager, and our program still functions as before. Let's create another class, and now for our CEO:

```
TCEO = class(TManager)
  public
    procedure AssignManager(const AEmployee: TEmployee);
override;
    procedure ReviewCompany;
  end;
```

and the actual implementation:

```
{ TCEO }
procedure TCEO.AssignManager(const AEmployee: TEmployee);
begin
  raise Exception.Create('The CEO can''t have a manager!');
end;
procedure TCEO.ReviewCompany;
begin
  // Do the company review
end;
```

Our TCEO overrides from the TManager, implements a new procedure ReviewCompany, and overrides the AssignManager, as this won't make sense to have a manager for a CEO. But now we have a problem. Let's say we change our implementation to an instance of TCEO:

```
var
  Employee: TEmployee;
  Manager: TEmployee;
begin
  Manager :=  TEmployee.Create;
  Manager.Name :=  'Jane Smith';
  Manager.Salary :=  31000;

  Employee :=  TCEO.Create;
  Employee.Name :=  'John Smith';
  Employee.AssignManager(Manager);
  Employee.Salary :=  22500;
end;
```

Although our project still compiles, we now have a problem when we run this program, because we suddenly get an exception on the AssignManager call. As this breaks our project, this is clearly a violation of the LSP. We should be able to change our TParent to TChild without having a side-effect on the functionality.

As you see, the implementation of the TCEO class is stricter than its parent class. The Liskov Substitution Principle states that you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your child classes.

Similar rules apply to the return value of a function. The return value of a function of the child class needs to comply with the same rules as the return value of the function of the parent.

So, how can we solve this?

The first thing is to ask ourselves: is a CEO really an Employee? Sort of; a CEO can have a salary, but to assign a manager won't make any sense. Let's try to solve this with the use of interfaces (again, program against interfaces, not implementation). We start with some interfaces:

```
type
  IBaseEmployee = interface
    procedure SetName(const Value: string);
    procedure SetSalary(const Value: Double);
    function GetName: string;
    function GetSalary: Double;

    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;
  end;

  IManagedEmployee = interface(IBaseEmployee)
    procedure AssignManager(const AEmployee: IBaseEmployee);
  end;

  IManager = interface(IBaseEmployee)
    procedure DoAppraisal(const AEmployee: IBaseEmployee);
  end;

  ICEO = interface(IManager)
    procedure ReviewCompany;
  end;
```

As you can see, we have an interface for the basic properties of an employee; we have created a specific interface for the manager and the managed employee, and one for the CEO. The implementations of our classes are now as follows:

```
TBaseEmployee = class(TInterfacedObject, IBaseEmployee)
  strict private
    FName: string;
    FManager: IBaseEmployee;
    FSalary: Double;

    procedure SetName(const Value: string);
    procedure SetSalary(const Value: Double);

    function GetName: string;
    function GetSalary: Double;
  public
    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;
  end;

  TEmployee = class(TBaseEmployee, IManagedEmployee)
  strict private
    FManager: TEmployee;
  public
    procedure AssignManager(const AEmployee: IBaseEmployee);
  end;

  TManager = class(TEmployee, IManager)
  public
    procedure DoAppraisal(const AEmployee: IBaseEmployee);
  end;

  TCEO = class(TBaseEmployee, ICEO, IManager)
  public
    procedure ReviewCompany;
    procedure DoAppraisal(const AEmployee: IBaseEmployee);
  end;
```

And finally, the changed calls to these classes:

```
var
  Employee: IBaseEmployee;
  Manager: IBaseEmployee;
begin
  Manager :=  TManager.Create;
  Manager.Name :=  'Jane Smith';
  Manager.Salary :=  31000;

  Employee :=  TCEO.Create;
  Employee.Name :=  'John Smith';
  Employee.AssignManager(Manager);
  Employee.Salary :=  22500;
end;
```

If you take a good look at the implementation, you can spot an error: the last Employee.AssignManager will give a compile error, because the TCEO class doesn't have the AssignManager function anymore. So, our logic must change, reflecting the actual situation.

And although we do use inheritance with our classes, we still program against interfaces, as you can see in another code example below:

```
var
  CEO: ICEO;
  Manager: IManagedEmployee;
begin
  CEO :=  TCEO.Create;
  CEO.Name :=  'John Smith';
  CEO.Salary :=  75000;

  CEO.ReviewCompany;

  Manager :=  TManager.Create;
  Manager.Name :=  'Jane Smith';
  Manager.Salary :=  31000;
  Manager.AssignManager(CEO);
end;
```

The implementation of our classes is now also compliant with the Liskov Substitution Principle, as we can now swap the TManager.Create with a TEmployee.Create without affecting runtime behaviour.

So, to summarise the Liskov Substitution Principle again: If TChild is of a subtype of TParent, then objects of type TParent may be replaced with objects of type TChild, without breaking the logic of the program. Don't enforce stricter behaviour in your child classes than what pertains in your parent classes.

INTERFACE SEGREGATION

SOLID principle 4: Interface Segregation. "Clients should not be forced to depend upon interfaces that they do not use."

The interface segregation principle is about finding the most appropriate abstractions in your code. Wikipedia has a nice description of this principle: "ISP splits interfaces that are very large into smaller and more specific ones, so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces."

Let's have a look at our previous Delphi interfaces from the Liskov Substitution Principle:

```
type
  IBaseEmployee = interface
<getters and setters>
    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;
  end;

  IManagedEmployee = interface(IBaseEmployee)
    procedure AssignManager(const AEmployee: IBaseEmployee);
  end;

  IManager = interface(IBaseEmployee)
    procedure DoAppraisal(const AEmployee: IBaseEmployee);
  end;
```

```
ICEO = interface(IManager)
    procedure ReviewCompany;
  end;
```

As you know, projects that are maintained and expanded over a long period of time inevitably receive requests that need to be implemented. Suppose you get a request to also store hired employees, who work for an external company plus the cost of hiring. The simplest way of solving this would be to simply create two new fields in the IBaseEmployee: “HiringCosts” and “Company” of type “TCompany”. In this way, you still comply with the Liskov Substitution Principle, because you can simply replace an IBaseEmployee with an IManager, and all the code will continue to work functionally as well.

```
IBaseEmployee = interface
{<'Implementations'>}
    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;

    property HiringCosts: Double read GetHiringCosts write
SetHiringCosts;
    property Company: TCompany read GetCompany write
SetCompany;
  end;
```

My experience is that you see this happen a lot in projects that have been around for a long time. It is in fact the quickest way to simply place the new functionality in the basic interface, or one of the derived interfaces. This makes the new functionality immediately available at the place in the implementation where you need it. It also complies with the Single Responsibility Principle; from the perspective of an HR employee, this functionality also belongs in the IBaseEmployee interface.

However, now the Interface Segregation Principle comes into play. The risk of this is that you not only make the interface (and therefore the class itself) larger and larger, but you also introduce functionality where it is not needed at all.

If we think about the right level of abstraction, it makes much more sense to put separate functionality for the hired workers in a separate interface. And while we’re at it, why not modify the Manager interface as well? If we look at the ISP, a Manageable employee is actually a separate role.

If we go through all this, we end up with the following interfaces:

```
IBaseEmployee = interface
  <getters and setters>
    property EmployeeID: Integer read GetEmployeeID write
SetEmployeeID;
    property Name: string read GetName write SetName;
    property Salary: Double read GetSalary write SetSalary;
  end;

  IHireable = interface
  <getters and setters>
    property HiringCosts: Double read GetHiringCosts write
SetHiringCosts;
    property Company: TCompany read GetCompany write
SetCompany;
  end;

  IManageable = interface
    procedure AssignManager(const AEmployee: IBaseEmployee);
  end;

  IManaging = interface
    procedure DoAppraisal(const AEmployee: IBaseEmployee);
  end;

  ICEO = interface
    procedure ReviewCompany;
  end;
```

Because we have moved the specific functionality to a role interface, our classes should reflect these changes. We have also added the TExternalEmployee. Shown are just the class headers:

```
TBaseEmployee = class(TInterfacedObject, IBaseEmployee)
<...>
  TExternalEmployee = class(TBaseEmployee, IHireable)
<...>
```

```
  TManager = class(TBaseEmployee, IManaging)
<…>
  TCEO = class(TBaseEmployee, ICEO, IManaging)
<…>
```

Have your interfaces and classes designed like this, and you can use the functionality using the interface support call:

```
var
  Employee: IBaseEmployee;
begin
  Employee :=  TExternalEmployee.Create;
  Employee.Name :=  'Jack Smith';

  if Supports(Employee, IHireable) then
    (Employee as IHireable).HiringCosts := 3750;
```

Using the Interface Segregation Principle makes your interfaces small, manageable, and easy to use. If you want to hire a CEO, just add the IHireable to the class and you are good to go. In other parts of your application, let's say where you want to calculate total hiring costs, you only have to provide a list of IHireable items to do that (TList<IHireable>). You don't have to know what type of employee it is (or even if it is an employee. It could be of a completely different type if you want).

So, you might wonder: what is the difference between the Single Responsibility Principle and the Interface Segregation Principle? I think they are different sides of the same coin; the SRP looks at a class or interface from a design perspective, where you would group together the things that change for the same reason and separate the ones that differ. ISP looks at a class or interface from the user's or consumer's perspective; you should only see what you actually need. One last example to show the difference:

SRP valid, but ISP invalid:

```
IUserManagement = interface
    function GetUsers: TList<TUser>;
    function SaveUser(AUser: TUser);
  end;
```

ISP valid, but SRP invalid:

```
IObjectManagement = interface
    function SaveUser(AUser: TUser);
    function SaveProduct(AProduct: TProduct);
  end;
```

ISP and SRP valid:

```
IUserProvider = interface
    function GetUsers: TList<TUser>;
  end;

  IUserSaver = interface
    function SaveUser(AUser: TUser);
  end;
```

(And the same for the TProduct of course).

DEPENDENCY INVERSION PRINCIPLE

It's time for the last SOLID principle: the Dependency Inversion Principle:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

The general idea of this principle is simple: if high-level modules depend on low-level modules, we should change (invert) that, and both high-level and low-level modules should use abstractions, not implementations.

Let's start with the difference between high-level modules and low-level modules. High-level modules are modules that provide or use complex logic and are using other modules or classes. Low-level modules are more like 'utility' modules, not depending on or using other modules. Have a look at the following examples of low-level and high-level code:

LOW-LEVEL EXAMPLES

```
type
  TLogger = class
  //<…>
  Public
    //<…>
    procedure LogMessage(const AMessage: string);
  end;

implementation

{ TLogger }

procedure TLogger.LogMessage(const AMessage: string);
begin
  Writeln(AMessage);
end;
```

or

```
type
  TEmailer = class
  //<…>
  Public
    //<…>
    procedure SendMessage(const AMessage, ARecipient: string);
  end;

implementation

{ TEmailer }

procedure TEmailer.SendMessage(const AMessage, ARecipient:
string);
begin
  Writeln('Dummy implementation of email, just sent ' +
AMessage + ' to ' + ARecipient);
end;
```

HIGH-LEVEL EXAMPLES

```
type
  TTaskItem = class
  private
    FName: string;
    FDueDate: TDateTime;
    FIsFinished: Boolean;
    FUsername: string;
    procedure SetDueDate(const Value: TDateTime);
    procedure SetName(const Value: string);
    procedure SetUsername(const Value: string);
  public
    constructor Create;
    procedure CompleteTask;
    property Name: string read FName write SetName;
```

```
property Username: string read FUsername write SetUsername;
    property DueDate: TDateTime read FDueDate write
SetDueDate;
    property IsFinished: Boolean read FIsFinished;
  end;

  TTaskManager = class
  strict private
    FTasks: TList<TTaskItem>;
  public
    constructor Create;
    destructor Destroy; override;

    procedure AddTask(const ATask: TTaskItem);
    procedure CompleteTask(ATask: TTaskItem);
  end;
```

An example implementation of the TTaskManager class can be:

```
procedure TTaskManager.AddTask(const ATask: TTaskItem);
var
  Emailer: TEmailer;
begin
  FTasks.Add(ATask);

  Emailer :=  TEmailer.Create;
  try
    Emailer.SendMessage('Task ' + ATask.Name + ' added',
ATask.Username);
  finally
    Emailer.Free;
  end;
end;
```

As you can see in this example, our high-level module TTaskManager depends on the low-level module TEmailer (because of the TEmailer.Create call). And that is a problem, so we should invert that dependency. Why? You'll figure that out later on.

Both low-level and high-level modules should also depend on abstractions. How do we change this? The simplest way is to use interfaces. An interface is, by definition, an abstraction, so by using interfaces instead of classes, we remove the dependency on implementations. As an extra benefit, we meet the second requirement of the Dependency Inversion as well, because an interface will not have the knowledge of the details involved of how things get done in the actual implementation.

Since we are concentrating here on explaining the fifth Solid Principle, I will not explain in detail how to add an interface to a class and how to use it. In the previous posts about the Solid principles, we have done that several times. In the end, we will have the following interfaces:

```
ILogger
IMessageSender
ITaskItem
ITaskManager
```

The implementation of classes becomes as follows:

```
TLogger = class(TInterfacedObject, ILogger)
TEmailer = class(TInterfacedObject, IMessageSender)
TTaskItem = class(TInterfacedObject, ITaskItem)
TTaskManager = class(TInterfacedObject, ITaskManager)
```

Okay, so let's start our transition to meet the Dependency Inversion Principle by changing this example of the TTaskManager. There is more than one way of doing this. One is by using Dependency Injection. Dependency Inversion is the principle, and Dependency Injection is a way of making this principle work. However, it's not mandatory to use Dependency Injection. Let me explain this.

Think of the dependency of our TTaskManager on TEmailer. Where does this dependency actually happen? The easiest way of identifying dependencies is by looking for .Create calls. In this case, the line

```
Emailer :=  TEmailer.Create;
```

introduces a dependency on the TEmailer class. In essence, every .Create call is a dependency on another class. As we have said, we do not want high-level modules to depend on low-level modules. So, for this class, we can use Dependency Injection to solve this. It is very simple: just inject the dependency via the constructor of the TTaskManager class:

```
constructor TTaskManager.Create(AMessageSender:
IMessageSender);
begin
  FMessageSender :=  AMessageSender;
  FTasks :=  TList<ITaskItem>.Create;
end;
```

We now have injected the dependency on the emailer (or actually, to the IMessageSender) class, so we can use it without any problem:

```
procedure TTaskManager.AddTask(const ATask: ITaskItem);
begin
  FTasks.Add(ATask);
 FMessageSender.SendMessage('Task ' + ATask.Name + ' added',
ATask.Username);
end;
```

And that is just how dependency injection works. As long as you consequently push the creation of classes back (i.e., do not use the .Create anywhere) you will automatically use Dependency Injection.

But where does this lead us? In the end you have to create a class somewhere, right?

If you push back the creation of classes, you will end up in the program files (the DPR). Have a look at my example for this TaskManager example program:

```
program Solid5;
{$APPTYPE CONSOLE}
uses
//<…>

var
  ATask: ITaskItem;
  ATaskManager: ITaskManager;
  AMessageSender: IMessageSender;
begin
  AMessageSender :=  TEmail.Create;

  ATaskManager :=  TTaskManager.Create(AMessageSender);

  ATask :=  TTaskItem.Create;
  ATask.Name :=  'Develop module X';
  ATask.UserName :=  'Marco Geuze';
  ATask.DueDate :=  Tomorrow;
  ATaskManager.AddTask(ATask);

  ATask :=  TTaskItem.Create;
  ATask.Name :=  'Codereview module Y';
  ATask.UserName :=  'Marco Geuze';
  ATask.DueDate :=  IncWeek(now, 2);
  ATaskManager.AddTask(ATask);

  ATask.CompleteTask;

  Readln;
end.
```

In the example code, we create two tasks, add the tasks to the Taskmanager and complete one task. We create only classes in the DPR right now. But in fact, isn't the DPR itself also a high-level module? I think it is, and again, we should not create a dependency in a high-level module.

I've shown you the way of doing this in classes via Dependency Injection. Here I'd like to show you how it's done via a factory class. The basic role of a factory class is to provide instances of interfaces. And as we are at the top level of our program, this is the place where we will create our classes. Let's create a new unit and create our factory class in this new unit. As this is a factory class, we'll use class functions to give back instances of classes:

```
unit ClassFactory;

interface

uses
//<…>

type
  TClassFactory = class
  public
    class function CreateTaskManager: ITaskManager;
    class function CreateTask: ITaskItem;
    class function CreateLogger: ILogger;
    class function CreateMessageSender: IMessageSender;
  end;
```

Let's look at the implementation of the CreateTaskManager function for example:

```
class function TClassFactory.CreateTaskManager: ITaskManager;
begin
  Result :=  TTaskManager.Create(CreateMessageSender);
end;
```

and for the CreateMessageSender function:

```
class function TClassFactory.CreateMessageSender:
IMessageSender;
```

```
begin
  Result :=  TEmailer.Create;
end;
```

and lastly to the DPR file again:

```
program Solid5;
{$APPTYPE CONSOLE}
uses
//<…>

var
  ATask: ITaskItem;
  ATaskManager: ITaskManager;
begin
  ATaskManager :=  TClassFactory.CreateTaskManager;

  ATask :=  TClassFactory.CreateTask;
  ATask.Name :=  'Develop module X';
  ATask.UserName :=  'Marco Geuze';
  ATask.DueDate :=  Tomorrow;
  ATaskManager.AddTask(ATask);

  ATask :=  TClassFactory.CreateTask;
  ATask.Name :=  'Codereview module Y';
  ATask.UserName :=  'Marco Geuze';
  ATask.DueDate :=  IncWeek(now, 2);
  ATaskManager.AddTask(ATask);

  ATask.CompleteTask;

  Readln;
end.
```

To summarise what we have done so far: in order to comply with the fifth Solid principle, we first use dependency injection to bring up the dependencies as far as possible. Because of this, no create call is used in the classes anywhere. Secondly, we converted the code to interfaces, so that we are no longer dependent on details, but on abstractions. And finally, we created one factory class in which all objects are created.

Finally, let's see what advantage this gives us. In the code above, we have an Emailer class that takes care of sending an email when a task is completed. Now suppose we don't want to do this via email, but via a text message. In the old situation we would have had to change the TEmailer instance everywhere it's used to TTexter. This is because high-level modules were dependent on these low-level modules. In the new structure, it is very easy to change this. We only need to change the CreateMessageSender class in the ClassFactory:

```
class function TClassFactory.CreateMessageSender:
IMessageSender;
begin
  Result :=  TTexter.Create;
end;
```

and with just one line of code, the whole application is adapted to the new requirements. Taking the other Solid principles into account (Liskov Substitution, Open-Closed), we can make this adjustment without any problems. But there are more benefits. Let's say we want to add unit tests to our application. If you just create a unit test project and test this application 50 times a day, you will end up with 50 emails, or 50 text messages in your inbox. So, in your unit test project, you can now simply have another factory class, with a dummy or mock implementation of the MessageSender class. And that's all – no more emails or text messages.

So, now you know the five SOLID principles. But how do you start? Just reading these principles alone won't work. You need to study your own code, find bad and good examples, and practise. Practise a lot.

- ☐ Start with one principle at a time, but build up your toolbox and learn all five.
- ☐ Read about the five Solid principles on our blog: https://gdksoftware.com/knowledgebase

CODING, THE DELPHI WAY

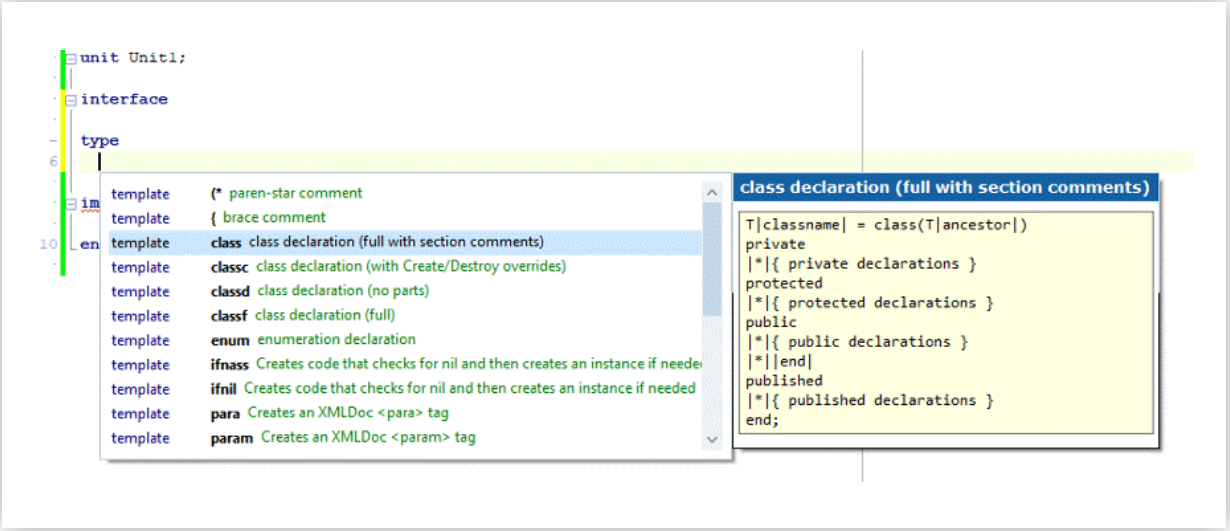
So far, we've covered some general tips, some programming methods, and development principles. Although they are tailored around Delphi, you can apply these to other languages as well. In this part, I will talk about some various specific Delphi-related coding guidelines and tips. In no particular order:

- Fix all hints and warnings. When you have done this in your existing projects, all new hints or warning will immediately catch your eye. I worked with projects that produced a couple of hundred warnings, and projects with none at all. In the latter case, working was so much more enjoyable and it gives you a more positive feeling and confidence to make changes in the source code.

- If you are working with objects, make sure to immediately write the code to free the objects. Or, even better, make use of interfaced objects, so you can make use of the garbage collector of Delphi.

- Use the Object Pascal coding guidelines: https://link.gdk.software/pascalguide. The introduction of the original version of this document expresses this nicely: "Object Pascal is a beautifully-designed language. One of its great virtues is its readability. These standards are designed to enhance that readability of Object Pascal code. When developers follow the simple conventions laid out in this guide, they will be promoting standards that benefit all Delphi developers by using a uniform style that is easy to read. Efforts to enforce these standards will increase the value of a developer's source code, particularly during maintenance and debugging cycles."

- Make use of the Live Templates (or Code Templates, as they were formerly called). Since Delphi 3, these templates are present in Delphi, but we still see that this handy feature is undervalued. Most developers are familiar with the code completing of if or while, where the structure of the statement is created automatically. But with Live Templates you can automate much more. If you want to see a list of the available Live Templates, you can use the shortcut Ctrl + J.



The more you can automate, the better. That's why it is recommended that you create your own live templates for frequently-used code within your program. This can easily be done via File -> New -> Other -> Code Template. For more information about the structure of the templates, go to the wiki of Embarcadero.

- Make use of the enormous existing libraries of Delphi. Almost everything you can think of has been written over the last 27 years of Delphi. Make use of lists like https://link.gdk.software/awesomepascal to find the right tool or library for the job.

- [ ] Fix all hints and warnings
- [ ] Check if you've eliminated all memory leaks
- [ ] Check the Object Pascal Coding guidelines
- [ ] Use Live Templates

NEXT STEPS

We have come to the end of this book. But hopefully, this is the beginning of your further development in the field of Delphi. As with many things in life, you will always be able to learn more and improve yourself. I hope I helped you a bit with this book. If you want to read further, I can recommend the following books:

Object Pascal Handbook (2021 edition) – Marco Cantú
https://link.gdk.software/objectpascalhandbook

Coding in Delphi – Nick Hodges
https://link.gdk.software/codingindelphi

More Coding in Delphi – Nick Hodges
https://link.gdk.software/morecodingindelphi

Dependency Injection in Delphi – Nick Hodges
https://link.gdk.software/dependencyinjection

Code Faster in Delphi – Alister Christie
https://link.gdk.software/codefasterindelphi

Thanks for reading – and happy programming!

Marco

Feedback, ideas, or tips?
Reach out to me via

marco@gdksoftware.com