TMS FixInsight
DEVELOPERS GUIDE

November 2017

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Installation and Uninstallation

FixInsight can be installed by:

1. Download the installer
2. Run the installer and follow the instructions

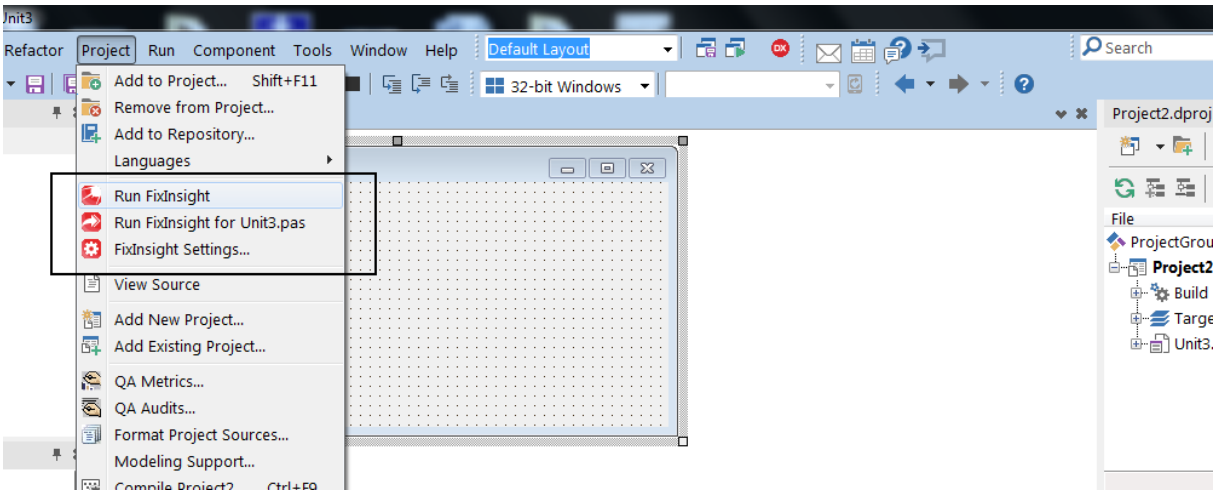
FixInsight can be uninstalled by:

1. Open Windows Control Panel
2. Select 'Uninstall a Program'
3. Find FixInsight in the list of programs and double click on it
4. Click 'Yes' button in the FixInsight Uninstall window

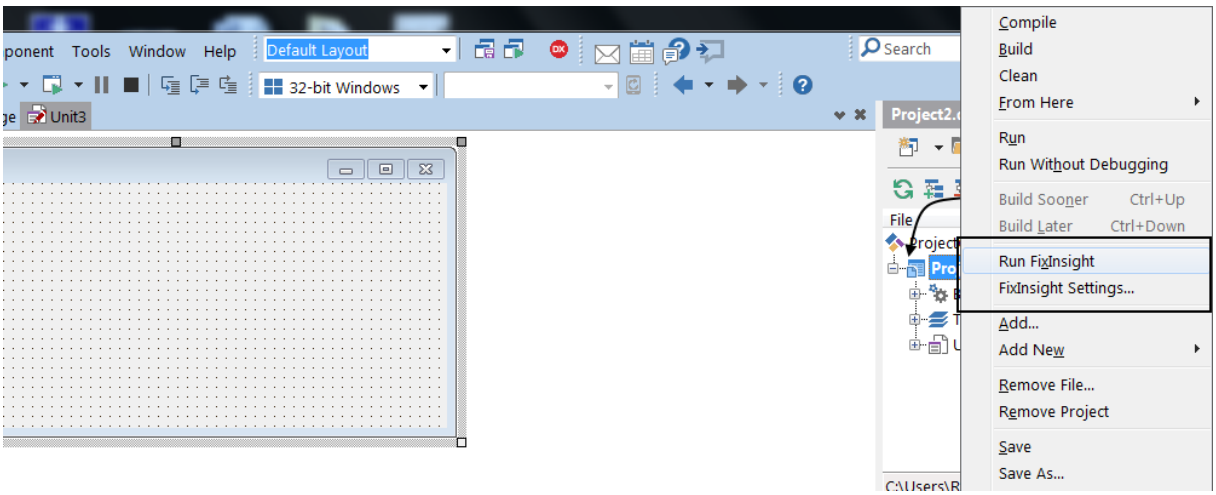
General Usage

FixInsight supports RAD Studio 10.2 Tokyo as well as the older IDE releases (2006 – 10.1 Berlin).

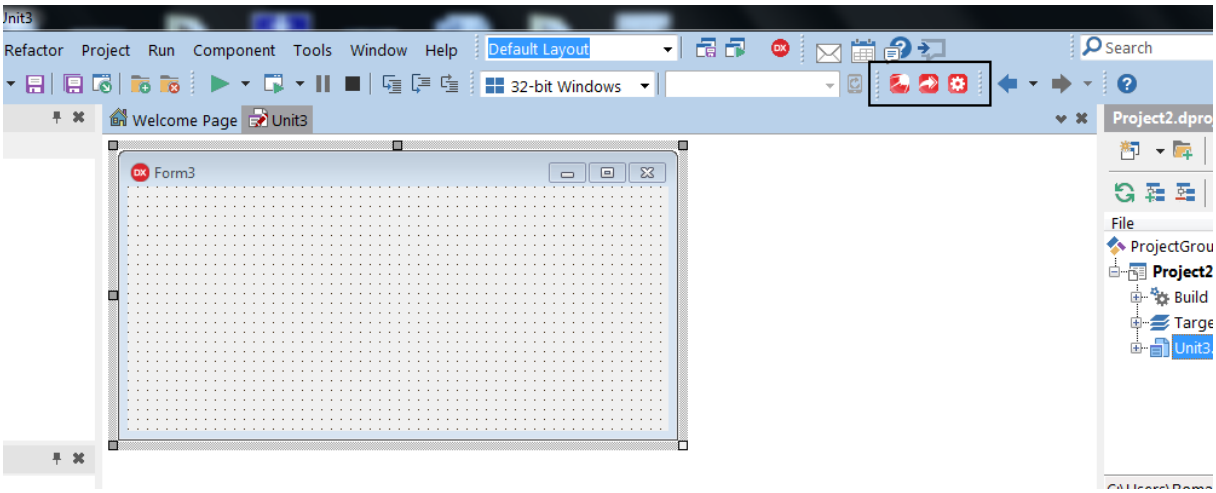
FixInsight adds menu items in the RAD Studio 'Project' menu:



In Delphi 2010 and above it also adds menu items in the project manager's context menu:

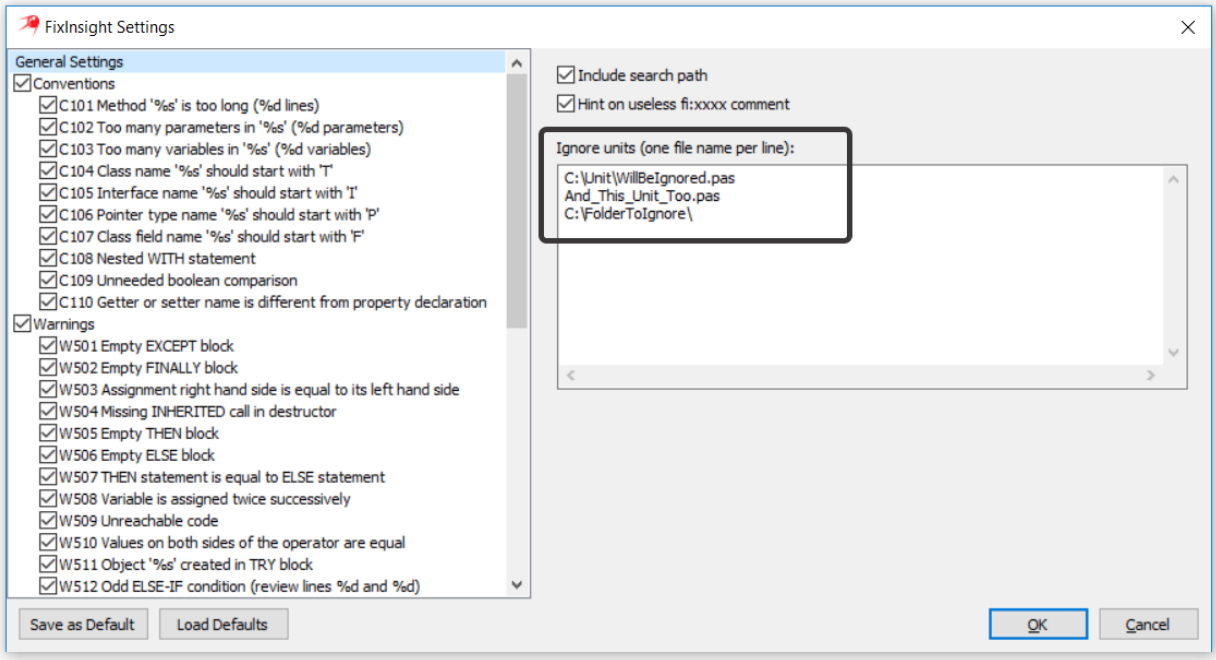


Click View->Toolbars->FixInsight to view the FixInsight toolbar:



Excluding Units from Analysis

You can exclude specific units from analysis by entering their names in the Settings window, one entry per line.



You can use file masks. Like “Jwa*.pas” or “C:\Project\ThirdParty*.pas”. Also you can ignore whole folder content: “C:\Project\FolderToIgnore”.

The second option is to add "FI:ignore" comment to the first line of a unit that you want to skip.

For instance:

```
unit Unit1; //FI:ignore
```

Suppressing warnings

You may use a special comment "FI:<RULENUMBER>" to suppress a particular warning. You need to place this comment on the line where a warning occurs.

For instance:

```
1    procedure RestartTimer;
2    begin
3      FTimer.Enabled := False;
4      FTimer.Enabled := True; //FI:W508 - this warning will be ignored.
5    end;
```

The second option is to use the _FIXINSIGHT_ compiler directive to suppress warnings. It works just like regular compiler conditional directives. Make sure that your code remains valid when the directive is disabled and the code segment is excluded.

For instance:

```
1    {$IFNDEF _FIXINSIGHT_}
2     procedure RestartTimer; // this code will be ignored
3     begin
4       FTimer.Enabled := False;
5       FTimer.Enabled := True;
6     end;
7    {$ENDIF}
```

By checking “Hint on useless fi:xxx comment” option in Settings window you can get a list of line where no warning produced by fi:xxx comment is present. This usually happens when you want to ignore a false positive but then you have installed a FixInsight update, where this false positive is fixed.

Command Line Tool

The FixInsight command line tool (FixInsightCL.exe) may be used in the build process or with continuous integration.

Parameters:

```
--project=XXX.dpr (a project to analyze)

--defines=XXX;YYY;ZZZ (compiler defines separated by semicolon)

--output=XXX (write output to a file)

--searchpath=XXX (unit directories)

--settings=XXX.ficfg (override project settings)

--silent (produce no output if no issues were found)

--xml (format output as xml)
```

A --project parameter is mandatory, other parameters are optional.

Usage example:

```
FixInsightCL --project=c:\source\example.dpr
```

Code patterns handled by FixInsight

The tool produces a list of warnings when issues are found. It also checks your code for coding convention compliance.

Warnings

Potential errors and oddities.

W501 Empty EXCEPT block

The empty except block warning means the exception is caught, but not handled correctly, which in turn means the cause of the exception that occurred in the try block is still there and affecting the app is executed. Want to produce a chain reaction of errors that are impossible to trace? An empty except block is a perfect way to achieve this.

```
1    try
2      Result := GetSomeData;
3    except
4      // nothing
5    end;
```

W502 Empty FINALLY block

The finally block allows your app to free memory and finish all running process correctly. The empty finally block means the app doesn't care much about releasing handles, freeing memory and finalizing objects and therefore becomes a fertile ground for mystical elusive bugs.

```
1    try
2      Result := GetSomeData;
3    finally
4      // nothing
5    end;
```

W503 Assignment right hand side is equal to its left hand side

While such statements do not affect performance or produce errors, they are potentially dangerous if a different expression was expected on the right hand side. This type of assignment is commonly seen with copy-pasted code.

```
1    RowCount := RowCount;
```

W504 Missing INHERITED call in destructor

Forgetting to call an inherited destructor means the object may not destroy completely since ancestor classes' destructors are not called, resulting in memory and resource leaks at best and unexpected behaviour at worst.

```
1    destructor TDataContainer.Destroy;
2    begin
3      FData.Free;
4    end;
```

W505 Empty THEN block

One semicolon renders the entire if clause useless. The DoSomething block always executes.

```
1    if Param > 3 then;
2    begin
3      DoSomething;
4    end;
```

W506 Empty ELSE block

Similar to the above, the else block executes regardless of the value of Param. The problem with such errors is that they are extremely difficult to track down. A semicolon is easy to miss.

```
1    if Param > 3 then
2      DoSomething
3    else;
4    begin
5      Flag := -1;
6      ShowMessage('Wrong param value');
7    end;
```

W507 THEN statement is equal to ELSE statement

Not a big deal if you did this intentionally (such as for debugging purposes). The check for Param effectively checks nothing and the same code is executed regardless of the check. This may indicate a logic error – that the two sides of the if/else should be different – or a copy/paste error.

```
1    if Param > 3 then
2      DoSomething
3    else
4      DoSomething;
```

W508 Variable is assigned twice successively

A simple error that is hard to track. Simply put, Value loses its value. Note, that the two lines of code below may be separated by 200+ lines of statements making detection of this potentially bug even harder.

```
1    Value := GetValue;
2    Value := 505;
```

W509 Unreachable code

The last line of this code block never executes. The Exit clause may reside deep in the logic of this block, so the problem becomes less evident than it is in the below example.

```
1    begin
2      Value := GetValue;
3      Exit;
4      ProcessValue(Value);
5    end;
```

W510 Values on both sides of the operator are equal

Copy-pasting of code fragments often helps, but sometimes they can become the source of undetectable issues. Here, both operands are the same resulting in 0 being assigned to Result.

```
1    Result := X - X;
```

W511 Object 'Foo' created in TRY block

A common mistake by those new to the language. When an object is created inside a try block, and the application raises an exception before the object instance is assigned to the Foo variable, such as if the constructor causes an error, Foo remains unassigned and the call to Free in the finally block will try to free a random pointer. This will cause a second exception.

```
1    try
2      Foo := TFooObject.Create;
3      Foo.CallMethod;
4    finally
5      Foo.Free;
6    end;
```

W512 Odd ELSE-IF condition

Why is this here? What does it do? Is the else part really needed? May be you know the answer, but more likely the values Param is checked against were different sometime in the past (or in

some other code you copied this fragment from). Now they are the same and can therefore be the source of errors.

```
1    if Param = 1 then
2      ProcA
3    else if Param = 1 then
4      ProcB;
```

W513 Format parameter count mismatch

The number of parameters passed to Format does not match the number in the format string. This will compile but will never work.

```
1    Result := Format('%s = %d', [Name]);
```

W514 Loop iterator could be out of range (missing -1?)

Dynamic arrays in Delphi are indexed from 0 to Length – 1, which seems to be incorrect in this code example.

```
1    var
2      I: Integer;
3      Arr: array of Char;
4    begin
5      Arr := GetArr;
6      for I := 0 to Length(Arr) do
7        ProcessChar(Arr[I]);
8    end;
```

W515 Suspect Free call

The call to Free looks suspicious in this context. For example, the code may have been intended to use ‘with’, but it is missing, or the entire block was copied from another part of code and then incorrectly modified. Either way, this warning indicates that the wrong object may be freed. In the below snippet, the THelloObject instance is being freed instead of the StringList instance.

```
1     procedure THelloObject.SaveToFile(const FileName: string);
2     var
3       StringList: TStringList;
4     begin
5       StringList := TStringList.Create;
6       try
7         StringList.Add('Hello world');
8         StringList.SaveToFile(FileName);
9       finally
10        Free;
11      end;
```

```
12    end;
```

W517 Variable 'Foo' hides a class field, method or property

See that local Foo variable in the GetFoo method? It hides the private class member of the same name, making that class field inaccessible from within the method. In the example below, the GetFoo method will always return the value of the local Foo, not the Foo field.

```
1     type
2       TMyClass = class
3       private
4         Foo: Integer;
5       public
6         function GetFoo: Integer;
7       end;
8
9     function TMyClass.GetFoo: Integer;
10    var
11      Foo: Integer;
12    begin
13      Result := Foo;
14    end;
```

W519 Method 'Foo' is empty

Empty methods are fine except they do nothing. Is this bad? Not unless you want the method to do something. Usually, empty methods are the remnants of past refactoring or changes in the class structure.

```
1     procedure TMyClass.Foo;
2     begin
3       // nothing
4     end;
```

W520 Parenthesis might be missing around IN operand

If Value is an integer it is calculated as "(not Value) in [set]", when usually what was meant is "not (Value in [set])".

```
1     procedure TMyClass.Foo;
2     var
3       Value: Integer;
4     begin
5       if not Value in [1,2,3] then
6         Bar;
7     end;
```

W521 Return value of function 'Foo' might be undefined

The Delphi compiler warns you if a function may not have had its Result initialised, but only if it is a simple / unmanaged type (eg an integer). However, it doesn't do this if it is a managed type, like an interface or string.

```
function TMyClass.Foo(Param: Boolean): string;
begin
  if Param then
    Result := 'OK'
  else
    DoNotAssignResult;
end;
```

W522 Destructor without an override directive

This directive must be set, or a call to the Free method will never be successful, because the Free method calls the destructor.

W523 Interface 'Foo' declared without a GUID

If an interface does not have a GUID, it cannot be used with Supports function or with the 'as' operator.

W524 Generic interface 'Foo' declared with a GUID

This means that any all generic instantiations of this generic interface will share the same GUID. Since interfaces must have a unique GUID, and GUIDs are used for casting, this can be an major error.

```
IFoo<T> = interface
  ['{E7B8DF46-3B3D-46D3-916A-6A6008DD5B68}']
  procedure DoWork;
end;
```

W525 Missing INHERITED call in constructor

The same as rule W504, but for constructors.

```
constructor TDataContainer.Create;
begin
  FData := TList.Create;
end;
```

W526 Pointer to a nested method

When a nested method is being called from outside its "parent" method, an incorrect stack frame often leads to an error that is hard to track down.

```
procedure TMyClass.DoWork;

  procedure NestedMethod(List: TObjectList);
  var
    I: Integer;
  begin
    for I := 0 to List.Count - 1 do
      Foo(Self, List[I]);
  end;

begin
  SetCallback(@NestedMethod);
end;
```

W527 Property is referenced directly in its getter or setter

Accessing a property in its getter or setter may lead to infinite recursion.

```
type
  TTestClass = class
  private
    FProp: Integer;
    procedure SetProp(const Value: Integer);
    function GetProp: Integer;
  published
    property Prop: Integer read GetProp write SetProp;
  end;

// ...

procedure TTestClass.SetProp(const Value: Integer);
begin
  Prop := Value; // cannot assign to Prop here
end;

function TTestClass.GetProp: Integer;
begin
  Result := Prop; // cannot get Prop value here
end;
```

W528 Loop variable is not used in FOR-loop

A loop variable is not used inside the loop. This may indicate a coding error.

```
for I := 0 to 9 do
  for J := 0 to 9 do
    Matrix[I, I] := 0; // J is not used
```

W529 Should be 'raise' instead of 'raise object'?

When we call 'raise E;' we are telling compiler that here is a new exception object that we want to raise. After Delphi raises the exception object, the original exception object is freed.

```
1   procedure TForm56.Button1Click(Sender: TObject);
2
3     procedure TestProc;
4     begin
5       try
6         raise Exception.Create('FIRST EXCEPT');
7       except
8         on e: Exception do // <-- Object 'e' will be released after executing this bloc
9         begin
10          ShowMessage('Except 1: ' + e.Message);
11          raise e; // <-- We should be doing this instead: 'raise;'. Don't reference 'E
12        end;
13      end;
14    end;
15
16  begin
17    try
18      TestProc;
19    except
20      on e: Exception do // <-- Object 'e' is already released in TestProc
21      begin
22        ShowMessage('Except 2: ' + e.Message);
23      end;
24    end;
25  end;
```

W529 Should be 'raise' instead of 'raise object'?

Interface GUIDs should not be duplicated across the project.

```
IMyInterface = interface
  ['{E9FFCD4E-E7B6-4B36-B4F3-A235CA926C17}']
end;

IAnotherInterface = interface
  ['{E9FFCD4E-E7B6-4B36-B4F3-A235CA926C17}']
end;
```

Optimizations

Little tricks that make your code cleaner and faster.

O801 CONST missing for unmodified string parameter 'Foo'

String parameter is not modified in a method. If you declare this parameter as a 'const' this will result in better performance. This will eliminate implicit try-finally and reference counting overhead.

```
1    procedure Output(S: string);
2    begin
3      ShowMessage('The message is "' + S + '"');
4    end;
```

O802 ResourceString 'Foo' is declared but never used

ResourceString is declared but never used. This means it may be removed without any risk.

O803 Constant 'Foo' is declared but never used

Constant is declared but never used. This means it may be removed without any risk.

O804 Method parameter "Foo" is declared but never used

```
1    function Output(A, B, C: Integer): Integer;
2    begin // Parameter C is never used and may be removed.
3      Result := A + B;
4    end;
```

O805 Inline marked routine "%s" comes after its call in the same unit

In order to be properly inlined an inline marked routine has to come before it's called in the same unit.

Coding Conventions

Coding convention violations are not technically incorrect but may slow down development or increase the risk of bugs in the future.

C101 Method 'Foo' is too long (N lines)

A method, function, or procedure that has grown too large.

C102 Too many parameters in 'Foo' (N parameters)

A long list of parameters in a procedure or function make readability and code quality worse.

C103 Too many variables in 'Foo' (N variables)

A long list of variables in a procedure or function make readability and code quality worse.

C104 Class name should start with 'T'

According to Object Pascal Style Guide class declaration should be prefaced by a capital T.

C105 Interface name should start with 'I'

According to Object Pascal Style Guide interface declaration should be prefaced by a capital I.

C106 Pointer type name should start with 'P'

According to Object Pascal Style Guide pointer type declaration should be prefaced by a capital P.

C107 Class field name should start with 'F'

According to Object Pascal Style Guide class field name should be prefaced by a capital F.

C108 Nested WITH statement

If a class's interface is changed, the behaviour of code using 'with' can change silently and without any indication. It creates imprecise semantics, and that is always bad.

```
1    with Obj1 do
2      with Obj2 do
3        DoSomething;
```

C109 Unneeded boolean comparison

Just an unnecessary comparison. Not a big deal, but a simple assignment operator looks more natural.

```
1    Result := A <> True;
```

C110 Getter or setter name is different from property declaration

Getter and setter method names are typically the name of the property with a 'Get' or 'Set' prefix.

```
1    property Caption: string read GetName write SetName;
```