# Maät Windows 8 coding guide

The current document gives an overview of hints and rules for coding the Windows 8 Maät application.

[Coding guidelines 2](#_Toc413613695)

[Project structure 2](#_Toc413613696)

[Clean class file 2](#_Toc413613697)

[Naming conventions 3](#_Toc413613698)

[Syntax 4](#_Toc413613699)

[Project Architecture 5](#_Toc413613700)

[Calidos.Maat.Windows 5](#_Toc413613701)

[Calidos.Maat.ClientLogic 5](#_Toc413613702)

[Calidos.Maat.ClientData 5](#_Toc413613703)

[Big picture 6](#_Toc413613704)

[Data structure 7](#_Toc413613705)

[Navigation 8](#_Toc413613706)

[No parameters 8](#_Toc413613707)

[Querystring as parameter 8](#_Toc413613708)

[QueryParameterDictionary as parameter 8](#_Toc413613709)

[Parsing navigation parameters 8](#_Toc413613710)

[Overview of pages 8](#_Toc413613711)

## Coding guidelines

To improve code readability between several developers, we try to stick to following coding guidelines. Other than these specific guidelines, common sense is a must in applying OO patterns, architectural decisions … If your class counts 1000+ lines or your method counts 100+ lines, they’re not doing a single thing and should be refactored into smaller pieces.

Don’t hesitate to question your thoughts and ask input from a colleague.

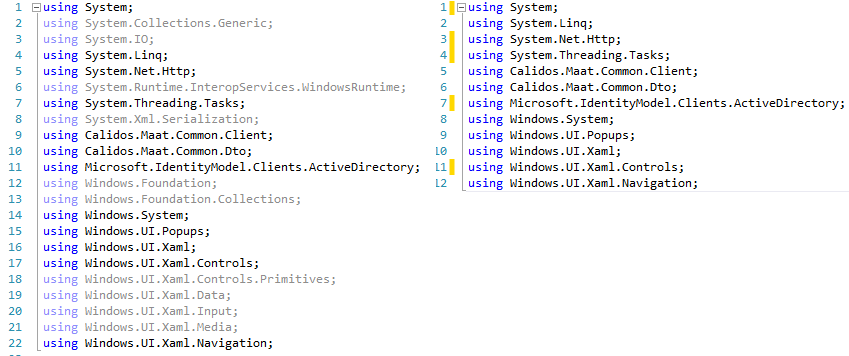
### Project structure

* One class or interface per file. Exceptions can be made for code generation and very small units that are considered part of one (e.g. event and event payload).
* The code file has at least the same name as the class in it. Extra parts can be added for clarity, e.g. xxRepository.GetDto.cs (all logic on getting).
* Subnamespaces are always represented by a correctly named folder. Note that folders can be used without being a namespace on itself. Note that VS automatically adds this namespace part, but several plugins (like ReSharper) warn for this incorrectly added part.

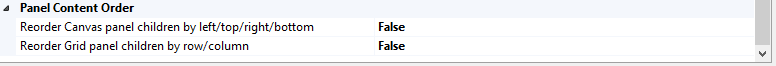
### Clean class file

The first major step to readable code is maintaining a clean class file.

* Remove unused code, big blocks of commented out ‘old’ code … If necessary, these can always be retrieved through TFS version control history.
* Try to clean up unused usings, either through an addin (e.g. ReSharper) or in Visual Studio itself. Note that you can bind any command (like remove unused usings) to a key combination. One forgotten using isn’t a disaster, 15 unused ones lowers readability.



* Use whitespace for code readability:
  + Between methods, properties …: one empty line
  + In code blocks add empty lines where necessary to ‘split logic’ for readability.
* Use the default code formatting (braces, spacing, 4 space indents …)
* Use XamlStyler VS plugin for XAML files with default values, except for Panel Content Order.



#### Class structure

Keeping the same structure in every class, will enable everyone to find easily find code without having to fall back to ‘dirty tricks’ like regions, comment blocks …

* Private readonly fields (mainly fields set by IoC in the constructor)
* Private fields, constants …
* Constructor
* Public fields
  + In case of a viewmodel:
    - bindable properties (hint: use code snippet to generate these)
    - bindable commands
* Public methods
  + In case of a viewmodel:
    - LoadDataAsync / SaveDataAsync
* Protected / private methods
* ‘Event handlers’, this also includes the callback method of a MVVM command.

### Naming conventions

#### XAML

* x:Name: XAML controls sometimes need a name (only when accessed from other XAML code), use full variable naming if you really need to know the type of the control or try to keep it out (e.g. CompactView can be GridView, ListView or even a set of random controls that is used for a compact representation).
* x:Uid: this field is used by the WindowsRT resources system for anything that can be set from a .resw/.xlf file. This is both localization and more advanced resource tricks on any UI element’s dependency properties (e.g. Width).  
  Prepend the key with the page’s name for page specific controls, this to make it easier to find back where a certain resource is used: e.g. x:Uid="LogEntry\_HeaderKeyFields".

The name of resources should show what type of resource it is:

* Converter: x:Key="VisibilityConverter"
* Brush: x:Key="StaticHeaderBrush"
* Margin: x:Key="GridGroupRightMargin"
* ItemTemplate: x:Key="LogTileDataTemplate" or x:Key="LogTileItemTemplate"
* Styles: show what the style can be used for in the name of the style
  + x:Key="TileTextStyle" TargetType="TextBlock"
  + x:Key="NavBarButtonStyle" TargetType="Button"
  + x:Key="NoHoverGridViewStyle" TargetType="GridView"

#### C# code

Don’t use Hungarian notation: ~~txtSomeText, iSomeInt~~ for variables. Give all variables a useful (**complete**) name. We don’t know what your ‘*lei’* is when we see code break at a certain line (where the variable clearly wasn’t declared). If it was named *logEntryInfo*, we know right away what we’re dealing with. After all, only the first time you need to put in the effort to type, rest is covered by IntelliSense.

* Private class fields: \_myfield;
* Variables somefield;
* Constant fields:
* Protected / public properties: PascalCased
* Methods / Classes:
* Although you should use complete variable names anywhere else, abbreviations are allowed for iterating (for/foreach: i/j/k …) and LINQ queries. But full names are still more readable.
* All async methods (returning Task or Task<T>) should end with –Async.

### Syntax

Newer versions of C# / .NET bring new code syntax, use them wisely but most important of all make sure your code stays readable. Also know which impact certain solutions have on your code (mainly performance), e.g. string concatenation with + or using a *StringBuilder*.

* *var* keyword: use when the type is clear from initialization. Use the type if it isn’t.
  + var myNumber = 1; var person = GetPerson();
  + int resultsSoFar = myCounter.ResultsSoFar();
* Always *await* an async method (or put compiler directives if it shouldn’t be awaited for a clear reason). Don’t return null in async methods, as awaiting null will crash your app.
* Try to avoid *async void*, only use this for top level even handlers. More info: [..\..\..\Phebus\InternalDoc\Technical Library\W8 - TL - Async await.docx](../../../Phebus/InternalDoc/Technical%20Library/W8%20-%20TL%20-%20Async%20await.docx)

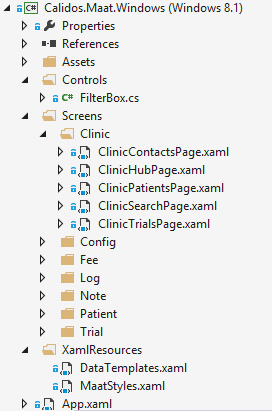
## Project Architecture

An overview of the projects in the solution:

### Calidos.Maat.Windows

The Windows 8 client is currently set up as a Windows 8.1 universal project, but only the Windows 8 part is implemented. Currently this project template exists out of a Windows, WindowsPhone and Shared part. These projects should only contain UI-specific content (pages, controls, styles, …). All logic should be placed in either the ClientLogic or the ClientData projects.

#### Pages

All pages should be placed under the Screens folder, and then grouped according to the context or application part they belong to. This structure is repeated as constant fields in the PageTokens class for navigation: Prism resolves both view and viewmodel types based on these tokens.

A page should inherit from Prism’s VisualStateAwarePage type and is automatically coupled with the view model through following attribute in XAML:

mvvm:ViewModelLocator.AutoWireViewModel="True"

#### Controls and usercontrols

(Templated) controls and user controls are placed in their own folders, either in the Windows project or the Shared project. It’s up to the developer to decide if further subfolders are useful or required to keep the project structure clean and maintainable.

#### Styles

All other general XAML resources (styles, converters, …) are located in the XamlResources folder. Try to keep to resource dictionaries clean and well-structured. Create a new dictionary to group certain styles/templates (e.g. everything config related). All shared styles/xaml resources (everything not declared on the page/control itself) should be documented in [CTO W8 - IG - UI styling.docx](CTO%20W8%20-%20IG%20-%20UI%20styling.docx). This prevents from a single “layout” (style) to be declared multiple times by multiple developers.

### Calidos.Maat.ClientLogic

All UI-related logic is placed in this project, an overview (not limited to these):

* Commands: we try to use the behavior SDK in combination with converters (see styles document), which limits the amount of custom commands to be created.
* Converters: see styles document.
* Events: events for Prism’s event aggregation.
* Models: models specific for the representation on a single view/control. These can aggregate dto’s from different object trees to simplify binding etc.
* ViewModels

### Calidos.Maat.ClientData

All data-related logic is placed in this project, an overview (not limited to these):

* Models: UI dto models (see data structure below)
* Repositories: single point of responsibility for data retrieval and transformation.
* WebApiFacades: mapping between REST api calls and strong typed C# code.

### Big picture

All projects mentioned above are tied together using Prism for Windows Runtime as MVVM framework and Autofac as IoC container. It speaks for itself that you should be familiar with following concepts (or ask a colleague to explain them):

* MVVM
* IoC and dependency injection
* XAML data binding, INotifyPropertyChanged (INPC), commands and behaviors

#### Autofac

The registration of types with Autofac for dependency resolving happens in the ApplicationBoostrapper class. Since we’re using an IoC container, we can register classes as singletons here as well, instead of implementing the singleton pattern in code.

#### Prism

First of all note that we’re trying to keep views and viewmodels completely separated. Do NOT declare UI controls in a viewmodel, and don’t hijack the view’s DataContext in code behind to access methods on the viewmodel if it can be solved in a clean way (data binding, behaviors and commands).

There are a few Prism specific implementations to keep in mind:

* A page uses an attached property to wire the viewmodel by convention (see above).
* Everything inheriting from BindableBase can use the SetProperty method for INPC.
* A viewmodel has 2 important methods in the lifetime of a page
  + LoadDataAsync: happens when the page is navigated to.
  + SaveDataAsync: happens when the page is navigated away from.

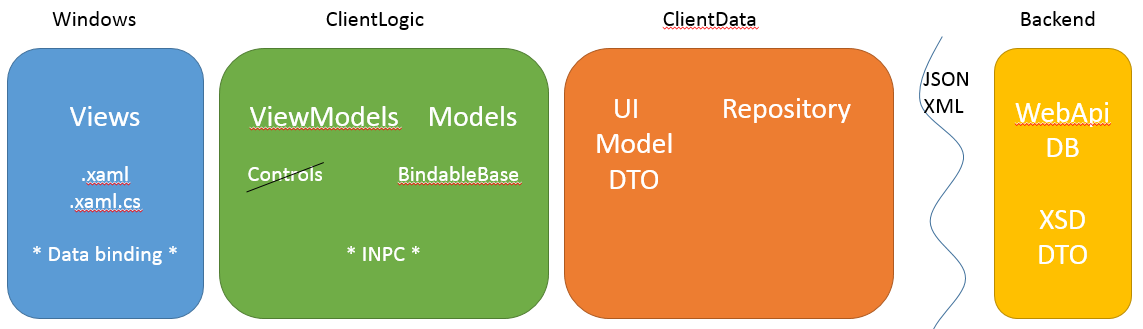
Note that between 2 pages, first the TO on page 2 happens and then the FROM on page 1.

* Navigation happens through the NavigationService on the viewmodel (see below).

More documentation and source can be found here: <https://prismwindowsruntime.codeplex.com/>

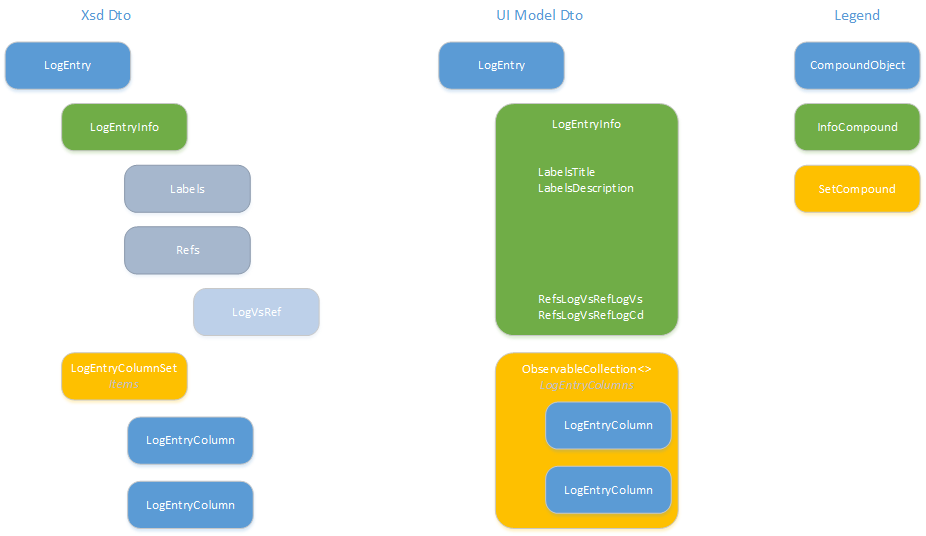
#### Program flow

When arriving on a page, the viewmodel’s LoadDataAsync is triggered. In this method, the viewmodel can retrieve all data it needs from the repositories (which he gets injected in the constructor). The repositories either get the data out of their cache or call the WebApi to retrieve data from the backend. It’s up to the viewmodel to bind either these data objects as-is, or transform them into a new model. The view should be agnostic of this process, as it retrieves everything through “some” data binding.



## Data structure

All data sent over the wire is formatted in the same tree-like structure to facilitate automation of certain tasks like in-memory merging. As this structure is too bloated to work easily with in the UI, a translation in both directions is done on the client side in the repositories. The translation can be seen on the picture below.



The root of every call is either a CompoundObject or an ObjectRef, which is a simplified object. Each object contains an InfoCompound and possibly a SetCompound (or SetRef).

The InfoCompound contains all fields of the object (see these as the fields stored in the same table) and possible reference keys to other objects). For the UI model, the structure of an InfoCompound object is flattened to one single layer by concatenation of property names, e.g. xx.Labels.Title becomes xx.LabelsTitle. This allows is to bind all properties on the same level, and keep change tracking of the InfoCompound in a single place. As soon as a single property is changed, the complete InfoCompound object is sent over the wire, as partial updates on this type are not allowed.

The SetCompound object contains some version fields and a single property Items, which is a collection of new CompoundObjects. Depending on whether AtVersion or SinceVersion is used, the complete or a partial set of sub-objects is sent over the wire (for more details see the WebApi document). In the UI model, the layer of SetCompound objects is removed and the parent object directly has a child property of the type ObservableCollection<T> with the child objects.

## Navigation

As we’re using MVVM, navigation should be triggered in the viewmodel, not in the view’s code behind. The *ViewModelBase* class has a property *NavigationService* that can be used by every derived view model. All existing page urls are defined in the *PageTokens* class.

### No parameters

NavigationService.Navigate(PageTokens.ConfigExam, null);

### Querystring as parameter

NavigationService.Navigate(PageTokens.ClinicHub, "id=-1");

### QueryParameterDictionary as parameter

A better way to pass parameters is using the *QueryParameterDictionary*. All supported parameter keys are defined in the *NavigationParameters* class. Simply add every parameter key-value pair to the dictionary and pass this are parameter in the *Navigate* method.

var parameters = CreateQueryParameterDictionary();

parameters.Add(NavigationParameters.Context, TempContext.Patient);

NavigationService.Navigate(PageTokens.NoteEntry, parameters);

### Parsing navigation parameters

When you arrive on a page, the complete navigation string including parameters is available in the deeplink variable. Retrieving the QueryParameterDictionary from this string can simply be done by calling the GetQueryStringParameters method, which is an extension method on string.

var queryParameters = deepLink.GetQueryStringParameters();

You can then either retrieve the values from this dictionary yourself or use the available extension methods to retrieve strongly typed values (or the default type value if the key isn’t specified).

private void ParseQueryParameters(QueryParameterDictionary queryParameters)

{

IsNew = queryParameters.GetBooleanValue(NavigationParameters.IsNew);

\_clinicCd = queryParameters.GetValue(NavigationParameters.ClinicCd);

\_logEntryId = queryParameters.GetIntValue(NavigationParameters.Id);

\_atVersion = queryParameters.GetDateTimeOffsetValue(NavigationParameters.AtVersion);

}

### Overview of pages

For a complete overview of all pages: check [CTO W8 - IG - Navigation.docx](CTO%20W8%20-%20IG%20-%20Navigation.docx).