Developers documentation Enigma Suite 2022.0.1

Jan Kampherbeek, January 2, 2022

Developers documentation Enigma Suite 2022.0.1

- Enigma Suite - introduction
- Choices made
  - Language: C# 10
  - Dependency Injection: Simple Injector
  - Testing
    - Unit tests: NUnit
    - Mocking: MOQ
  - Database: SQLite
- Coding conventions
- Technical aspects
  - Using the Swiss Ephemeris
- Architecture
  - Core
  - API
  - Handlers
  - Facades
- Astronomical aspects
  - School of Ram: hypothetical planets
  - School of Ram: oblique longitude

Enigma Suite - introduction

Enigma Suite is a software suite,written in C#.

The suite has 4 components:

- **Charts**: calculation and analysis of horoscopes.
- **Cycles**: calculation and presentation of astronomical cycles.
- **Counting**: support for astrological statistics.
- **Calculations**: astronomical calculations.

The program is free and open source.

This document provides information about the technical aspects for interested programmers.

Choices made

Language: C# 10

I will use C# 10 and .NET Core 6.0 from Microsoft.

After experimenting with Java, Kotlin, Free Pascal and Delphi I found that C# has the following advantages.

Compared to Java/Kotlin:

- Better language (compared to Java) or on equal foot (compared to Kotlin).
- UI Support. JavaFX is becoming risky and Jetpack Compose for Desktop is in its infancy. And as it is a Google product, Jetpack could be terminated at any moment. Using WPF has a lot of advantages: testability, good match for MVVM
- Deployment is integrated in Visual Studio (VS), no need for external tools to embed the runtime.
- VS is about as good as intelliJ.
- Documentation, less fragmented than with Java/Kotlin.
- Integration: a lot of functionality is directly available in the Community Edition of VS. For Java and Kotlin, your need to select many external libraries (sometimes frameworks). You can do the same in Visual Studio but it will often not be necessary.

Compared to Object Pascal (Free Pascal/Delphi):

- A very good language. Even if I prefer the Pascal syntax of Free Pascal (FP) and Delphi, I believe that C# does a better job regarding language structure. Especially if programming against interfaces, the existing interface section in FP/Delphi code is obsolete.
- Good support for the GUI, comparable to Delphi (FireMonkey) and better than Free Pascal. Supports a responsive UI.
- Available functionality like Dependency Injection and Mocking. FP does not support DI or mocking. For Delphi one good library (Spring4D) isn available, but there is only one maintainer. In general there are much more non-commercial libraries available if compared to FP/Delphi.
- Superior IDE, compared to Lazarus and also better than Delphi. About the same level as intelliJ.
- Documentation: more extensive and easily accessible.
- Long term availability. VS has been available in a free edition for a very long time, the community edition for Delphi at this moment for about three years. Free versions of Delphi have been discontinued in the past.

Of course, there are also some drawbacks:

- The use of .NET is required. This is not a big problem anymore for installations, as .NET Core can be included in the installer. But it seriously enlarges the footprint. This disadvantage is only compared to FP and Delphi, Java and Kotlin have the same problem.
- Performance could be less than the performance of FP or Delphi because of the use of the .NET environment. I intend to do some benchmarks.

The selection for C# is mainly based on the support for the GUI, the quality of the language and the documentation, and the expected long-term availability. What also counts is that I had previous positive experiences with C# (2008, using C# 3).

Dependency Injection: Simple Injector

All volatile objects will use DI. The approach is mostly based on *Dependency Injection: Principles, Practices, and Patterns* by Steven van Deursen and Mark Seeman. I will use a DI Container: Simple Injector, developed by Steven van Deursen. This DI Container performs well and has all required functionality. For a comparison with other frameworks, see: https://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison

Testing

Unit tests: NUnit

I will use nUnit for unit tests.
In Visual Studio the following solutions for unit testing are available.

- xUnit
- nUnit
- MSTest

MSTest is the standard solution from Microsoft but nUnit has more functionality and is a long time de facto standard. I did not investigate this thoroughly but after a short search on the Internet it appears that nUnit and MSTest are comparable. xUnit has some specifics, like a less understandable syntax. I will use nUnit, mainly because it is a long time proven solution.

Mocking: MOQ

For mocking MOQ is clearly the most used solution, so I will use this framework.

Database: SQLite

The data to save is mostly about data for charts and in a much smaller amount for configurations. A RDBMS is well suited to handle this type of data. As Enigma is a single-user application, concurrency is not a requirement but an embedded, zero-configuration, database is. SQLite is a perfect match. A simple but proven database engine that can easily be matched with C#.

Coding conventions

I will try to abide to the standards. For a definition check: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Technical aspects

Using the Swiss Ephemeris

For astronomical calculations, the Swiss Ephemeris (SE) is used. The Se consists of a set of data and a 64-bits dll: *swedll64.dll*.

To access the dll, the attribute [DllImport] is used. All imports from the dll are defined in facades (see the paragraph *Architecture*). As an example for the definitions I used the file swissdelphi.pas that Pierre Fontaine and others created to access the same dll from Delphi.

Architecture

The GUI is based on the MVVM model. The view consists of the XAML and accompanying C# part. An additional ViewModel is added to the view.

Core

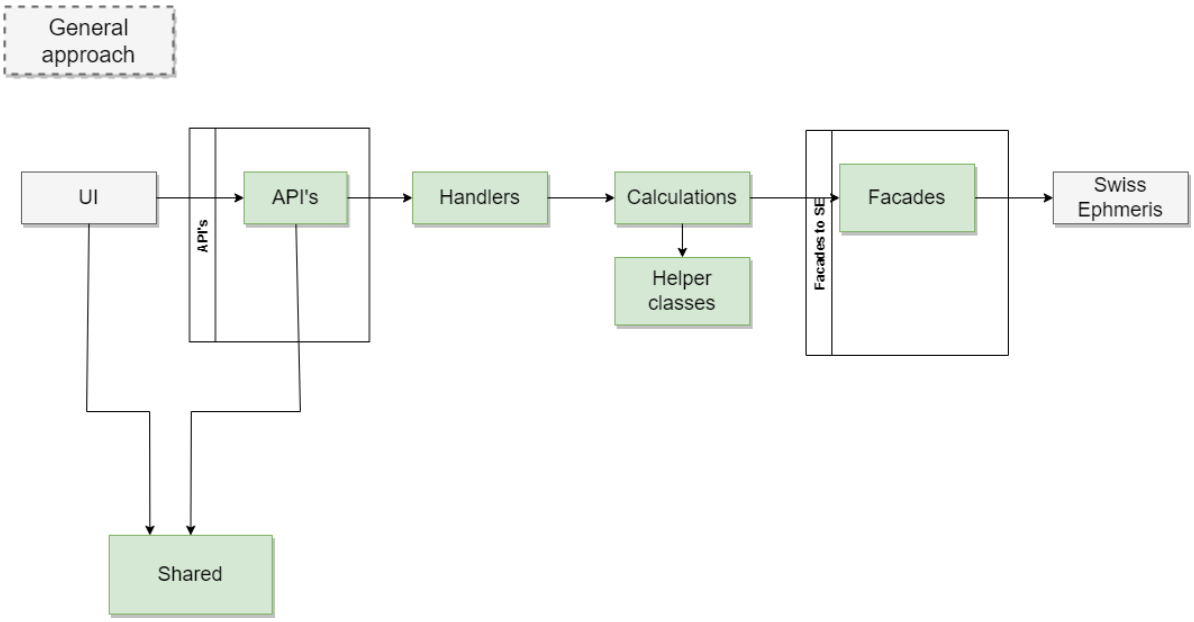
The core of the application handles all calculations and analyses. All requests should be done via an API. An API always uses a Handler to take care of the request. The Handler will use one or more other classes to perform the required actions and uses the results of these actions to return a response.

If the Swiss Ephemeris needs to be accessed, a separate Facade is called that is aware of the specifics of the SE.
All classes that are used by an API, except the Facades, are part of a functional whole, typically within one folder and one namespace. I use the term *swimming lane* for this functional whole.

The outside world of the Core will be the UI and external resources like the SE, databases etc.

Domain information is available in a part called *Shared* which is accessible for the UI and part of the core.

In the following diagram, all green boxes are part of the Core.



API

The API is divided in several groups: *AstronApi* for astronomical calculations, *DateTimeApi* for calculations related to clock and calendar etc. This will result in a relatively small set of classes, each with several public methods. These public methods provide the real API.

Each API method accepts an incoming request and, if the request is valid, will ask a *Handler* to take care of this request and return a response, which is returned to the caller of the API. To check the validity of a request, a set of guards is used in each API call.

Handlers

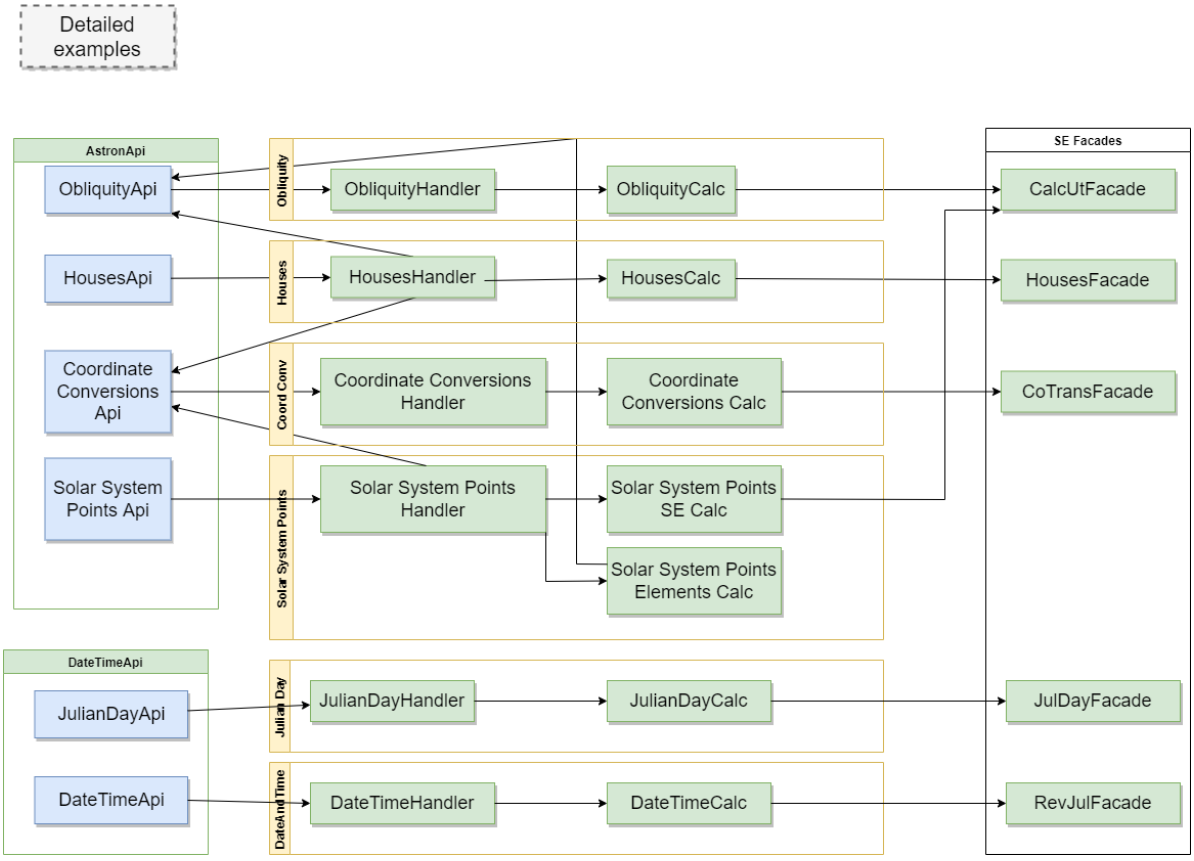
A handler is the starting point of a specific functionality. In the following diagram, each handler is shown in a kind of swimming lane. The swimming lane is typically implemented as a folder and a namespace for the same folder. Sub-folders and sub-namespaces are allowed.

A handler or other objects can access either objects in the same swimming lane, or API's in other swimming lanes. By accessing the API f another swimming lane, the validity of the request is ensured because of the guard statements in the API.

Facades

To access external functionality, a facade or comparable object is used. For the Swiss Ephemeris, this will be a facade (as shown in the following diagram). For a database, this will typically be a DAO.

The facades for the SE are outside of the swimming lanes as some facades (e.g. CalcUtFacade) will be accessed from several swimming lanes.



In the diagram classes are indicated by a green box. The green regions *AstronApi* and *DateTimeApi* also indicate classes. The blue boxes indicate a method.
The overview is schematic, eventually several classes will be added to some of the swimming lanes

Astronomical aspects

The usual approach using the Swiss Ephemeris is followed but some specifics need to be mentioned.

School of Ram: hypothetical planets

The three hypothetical planets as proposed by the School of Ram, Persephone, Hermes and Demeter, are supported.

The calculations are based on the orbitual elements and calculated separately, without accessing the SE.

School of Ram: oblique longitude

The School of Ram supports a solution for the projection of the solar system bodies to the ecliptic. This solution ensures a proper placing of bodies in a house. However, the projection to the ecliptic is skewed. The solution is called 'true place' and also 'astrological place'. I prefer the more correct term 'oblique longitude'.

A dedicated calculation of this oblique longitude is implemented in Enigma. Some background information will become available. [TODO].