Technical notes

Luis Rodríguez Neches

Everis

Contenido

[Analysis 2](#_Toc527892093)

[Assumptions 2](#_Toc527892094)

[Design 3](#_Toc527892095)

[Coding 5](#_Toc527892096)

[Testing 5](#_Toc527892097)

[Final notes 6](#_Toc527892098)

[References 7](#_Toc527892099)

# Analysis

I understood the problem of a star system as a mathematical problem with points in a circle, vectors and trigonometric calculations. Thus:

* Planets alignment and sun alignment can be considered as a vector problem of summing all the vector modules.
* Planets triangle containing the sun can be considered as a vector problem where another point is contained in the triangle formed by the other three points.

## Assumptions

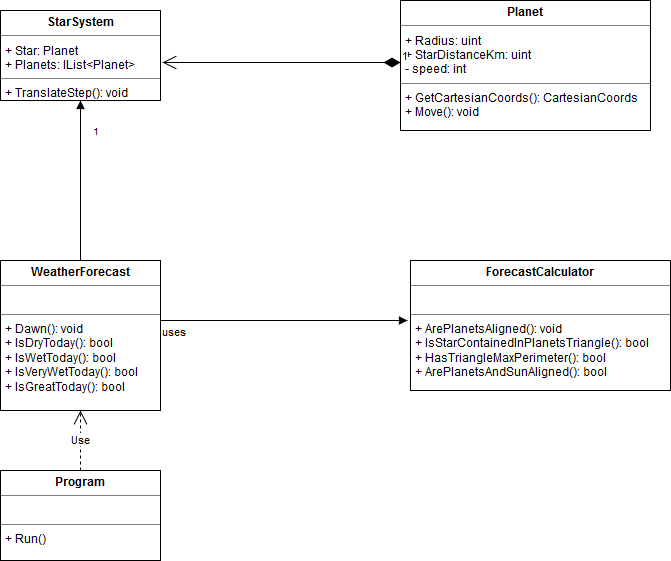
* The problem asks to make a prediction for 10 years, but we don’t really know if they are Earth years or one of the planets years. One of the planets has a year of 360 days, the other one of 120 days and the last one 72 days. In addition to that, Earth years have leap-years. Since this is not specified, I assumed 10 Earth, non-leap, consecutive years.
* The problem asks about periods, but it doesn’t specify if they are intended as consecutive days with the same weather conditions, so I will assume that by “period” it means just a “day”.

# Design

My principal objectives, taking into account that this is an algorithmic problem and not a real system were:

* KISS: keep the solution as simple as possible.
* SRP: every class must have only 1 responsibility.

The following is a UML class diagram, containing only the main classes:



* **Planet** represents a planet in the system. It can move and can retrieve its position in Cartesian coordinates.
* **StarSystem** is a Composite that represent a bunch of Planets that work together. The TranslateStep() allows, in parallel, moving the planets for 1 day.
* **WeatherForecast** implements the business logic of the system from a high abstraction level. It delegates in the ForecastCalculator the lower level calculations.
* **ForecastCalculator** implements the business logic of the system from a low abstraction level, including the mathematical calculations.
* **Program** is the entry point of the system.

By separating the logic between the WeatherForecast and the ForecastCalculator I accomplish the SRP principle, making it easier to reuse the ForecastCalculator or changing the WeatherForecast without affecting the calculations made.

The UML tries to represent the role of every class and their relationships. All the relationships could be represented as dependencies, but I wanted to make stress on the point that StarSystem is a composite of Planets, and that the WeatherForecast delegates in ForecastCalculator the hard job.

# Coding

I implemented the solution in .NET code because it is the language I’m more familiar with, and the one I feel more comfortable with.

I didn’t use a database to store the calculations to accomplish the KISS principle. The data is so simple that a database is pointless in this case. It adds complexity, maintenance, installation, configuration and performance bottlenecks.

By using a plain text file to store the predictions we make the installation of the application simpler (both the forecast app and the REST API app), and cross-platform.

The forecast app measures time to ensure that it does its job fast enough, including a means to ensure that the performance is good (less than 1 sec).

I try not to comment my code. Classes, methods and variables names should be good enough to explain purpose. Comments are hard to maintain and very often they lie. Code doesn’t.

Nevertheless I included comments in the most complex parts of the code, specially mathematical calculations, or if the purpose was not clear enough.

## Testing

I included unit tests for the system.

The framework used was MSTest. I’ve used NUnit and MSTest for years, and both are okay, but MSTest is included with VisualStudio, so it’s no needed to install a thing.

I didn’t use mock libraries because I didn’t need them.

Tests just cover the most important part of the system. I didn’t look for high-coverage, but to cover the most important, critical ones. Nowadays it is not as important high-coverage as good coverage: don’t cover trivial code snippets but cover the “meat”, the “stuff”.

When writing tests, sometimes I used the TDD approach, sometimes I didn’t. When it was clear to me the path to do the things, I used TDD. Regarding mathematical calculations I wrote tests after I wrote the code, or, sometimes I overlapped testing and coding.

In any case I consider unit testing as a coding process that cannot be separated from the code itself, and that’s why every piece of code I write contains unit tests, because I consider it part of my job.

# Final notes

I uploaded the code to GitHub:

<https://github.com/ravelus/planets>

and provided a build-all.cmd script to compile and run all the tests.

# References

I needed some help to remember trigonometric concepts. I found it useful the following web pages for this matter:

<http://www.dma.fi.upm.es/personal/mabellanas/tfcs/kirkpatrick/Aplicacion/algoritmos.htm#puntoInterior>

<http://funes.uniandes.edu.co/8137/1/pag2.html>