FROM LOW LEVEL TOOLBOX TO ORBIT DETERMINATION: HANDLING USERS REQUESTS IN OREKIT

L. Maisonobe, P. Parraud

CS-SI
Parc de la grande plaine - 5 rue Brindejonc des Moulinais
BP 15872
31506 Toulouse CEDEX 5
France

ABSTRACT

Orekit [1, 2] is a core space flight dynamics library published as free software under the terms of the permissive Apache Software License V2 [3]. Since its inception in 2002, Orekit was designed as a low level layer providing the foundation objects for operational applications. From the very beginning, the foundation objects included time, frames, orbits, but also attitude, maneuvers and a rich framework for state propagation with continuous output and on the fly detection of discrete events. As new versions were published, this set of features has been extended, with new propagators (the semi-analytical DSST being a major example), new predefined events (16 as of end 2015 and counting), new frames and various improvements.

As an open-source project, Orekit is essentially driven by its users requests and contributions. Looking back at the project evolution, one notices that after the initial stabilization phase during which the core features were completed, users requests led to introduce more and more intermediate features. These features where more mission or operation-oriented, showing us the library was used in various contexts for real problems solving. Many features added in the last two or three versions were really not envisaged at project start. These features clearly show the benefits we get from an open community. Some users have a problem to solve that first appears to be really mission-specific, but often as they notify the project about it, it appears more general than expected and can benefit other users after some reformulation. One typical example is the ellipsoid tessellation. This strange feature was needed for one project in early 2015, but just one month after its design, a second project raised a similar need and a few weeks later an independent user opened a feature request on the same topic.

Some other features have been on our plans for a few years without being implemented, both because of lacking resources to do the job and because they were considered at the boundary of Orekit scope. This was the status of orbit determination. Here again, as more and more users were demanding for it, we finally embarked on it and added it.

This presentation focus on how an open-source project can interact with its users while still maintaining a general orientation, using some examples from the last few releases of the library, up to the latest addition of orbit determination.

Index Terms— Open-source, Orekit, community

1. INTRODUCTION

Orekit [1, 2] is a free software library providing core space flight dynamics objects. It was started in 2002 and was made freely available under the terms of the Apache Software License V2 [3] since 2008.

2. A CONTINUOUS EVOLUTION

The history of Orekit starts in 2002, as CS decided to develop an in-house space flight dynamics product in order to be able to bid in international tenders. The decision was driven by too high licenses costs of available systems (regardless of them being agencies or commercial products) that penalized our offers. The development was slow as few resources were available and as everything was funded internally. As of 2006, an early version was available and technically exceeded the initial expectations, so it was considered worth to not only use it for bidding, but also to sell it as a product. This was a commercial failure. Two years later, all the space actors we approached declared the product was technically amazing and very well designed, but none wanted to get bound to yet another third-party controlled critical component.

CS decided therefore to open-source Orekit in 2008. The Apache Software License V2, a permissive license, was strategically selected [3, 4]. This license is often qualified as *business friendly*. These choices ruled out the concerns that restrained adoption of the library by giving control back to all potential users. This move was very well received by the space community, despite it surprised many people given the depth of the modelling and thorough design.

During the early years, despite Orekit was a free software library, its development was still performed in a *cathedral* mode [5]. This means all decisions and all development were done behind closed doors and users had to wait until a new version was published. Despite this closed process, we get the first external contributions. The process was improved and a public forge was opened in 2011 [6]. Users could participate more actively to the evolution of the library. The same year, the first external contributor was given direct commit access to the source repository.

The last milestone was reached in 2012: the governance of the project was also opened to other actors, using a model inspired by the Apache Software Foundation meritocracy [7]. Today, the Project Management Committee members come from agencies (ESOC, ESTEC, NRL), academics (ISAE, University at Buffalo), as well as private companies (TAS, Applied Defense Solutions, CS-SI).

The Orekit library is used by numerous actors and an ecosystem is gradually developing around it [8].

3. COMMUNITY

What are the key factors that changed a commercial failure into a successful project?

The first factor is the choice of the open-source model, and the choice of a *permissive* open-source license. Space systems are critical ones. Stakeholders are not ready to give up their control over such strategic components, even for the best designed library. Open-source gives this control back to them. Permissive licenses do not introduces fears to lose strategic advantages for the value-added components anyone can stack up on top of a low level library.

The second factor is the community. An open-source project without a community is a failed project. Lets look back more than 20 years ago, at an interesting example. While working in a public context, the main author did create an attitude simulation library. The library was innovative (and in fact would still be by todays standards). It had been operationally validated by several years of use in LEOP. This library was released as free software in order to foster its adoption by the space community, albeit under the terms of a strange home-grown license. However, no real effort was dedicated to create and animate a community. There was no time allocated for this and we hoped the technical features alone would draw crowds to it. They didn't. It was, and still is, a bitter disappointment, but it taught to some of us something valuable:

> Providing the best technical product, even free of charge, is not sufficient for success. You have to work with others, and care about them if you want them to adopt it.

This is a well known aspect of products development. It has been thoroughly analyzed a few years later by Eric S. Raymond in his renowned essay *the Cathedral and the Bazaar* [5]. The essay points out 19 lessons learned that influenced the open-source movement and are now widely adopted. Lesson number 7: "Release early, release often. And listen to your customers", is the most well-known quote from the essay.

Lesson number 6, despite less renowned, is also a key point. It appears in section *The importance of Having Users*. This lesson reads "Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging". We missed this element in the old attitude simulation library. It is the underlying theme of this paper.

When Orekit was published as a free software, we had a much better understanding of how open-source works, we had already learned from experience some (but not all) of the errors that should be avoided. Our goal was to really open as much as possible the development of the library and to adopt a bazaar-type development model. It was not possible from the very start, as the product was not yet established and setting up the infrastructure for collaborative required some work. Only a minimal static web site was set up and the Orekit team pushed versions as they were published.

4. INTERACTING WITH USERS

4.1. Point-to-point communication

During the initial years, as only a static web site was available, interactions were limited to a contact page that listed a few mail addresses for technical or administrative matters. The addresses were redirected internally to various people at CS-SI so they could answer questions. This means the communication was essentially point-top-point without any visibility. As space flight dynamics is not really a popular domain, this interaction was indeed low volume.

Despite these adverse conditions, the first interaction from a total outsider occurred less than one month after the first publication. We of course already had some discussions about the move to free software with our regular customers earlier, but this one was not on our radar. It started with questions (the first one concerning validation, as could be expected) and we replied as accurately and as thoroughly as we could, even not knowing who was our *first customer*, as my boss called him. A few weeks later, we got a follow-on of our first contact. This time, instead of questions he suggested some evolutions based on their use. The last sentence of this message was the one that really delighted us: "We know we can extend your base TLE class to accomplish this, but thought it might be something that would help others as well". They understood what open-source meant and complied to it. Of course, we did implement the feature they asked for (it was only a few lines of code) and it was published in the next version.

This type of interactions continued regularly and still happens today. Some random people jumping out of nowhere contact us saying they have been happy Orekit users for weeks, or months, and even years by now, and they have something to ask or to suggest, or even to contribute.

4.2. Collaborative tools

Point-to-point communication was only a temporary solution. It could not be used reliably for a full-fledged open-source project. The main drawback of this mode of communication is that it does not benefit the whole community. There are no public archives that can be indexed by search engines. This implies that the same question can be asked again and again as nobody can see that it has already been answered a number of times.

From the very beginning, the Orekit team wanted to have a full set of collaborative tools set up and publicly available. This was effective in 2011 with four main services started:

- public mailing lists with archiving,
- source code management repository with anonymous read access,
- software forge with issues tracker, activity, download area, wiki, connection to the Source Code Management System (SCM)
- static web site still available

All these services allowed to get rid of the ineffective cathedral development model and embrace the more efficient bazaar development model.

We selected a decentralized Source Code Management system: git. A decentralised SCM simplifies collaboration as external users can use it on their own even before they get write access to the official reference repository. They can keep their source tree synchronized with the upstream work. They can easily provide patches that they want to get included in the upstream code. Centralized source code management systems like subversion for example require more work for users. This is especially apparent when merging upstream work and own local work. Centralized SCM could be used, as numerous open-source projects prove it, but decentralized SCM are really a simplification for collaboration.

An important project management decision made at that time was that the official git repository available to external users *is* the repository used by the core Orekit developers themselves. It is not a clone lagging in time or containing only a subset of the features. The reasons for this choice are once again related to community management. A real time view of the commits made shows a more open policy. This also greatly speeds up issues corrections. When users open an issue ticket in the forge, for them it is important. With the development repository being public, they get the fix as soon as it is created by the development team. In some cases, we had fixes published as fast as 20 minutes after the issue has been notified. Doing this shows the users they are considered and it gives them more confidence about the project. This is of course not always possible and some issues remained opened for months or even years in some cases. These long delays correspond to issues that are either very difficult to reproduce (bad bugs reports) or solutions that need some incompatible changes that cannot be introduced between minor versions or are inconsistent with the project design decisions. Delays also happen for tickets that are not issues but are rather features requests that may be costly to develop and have to wait until they can be taken into account.

Just as collaborative tools were made available, the Orekit project welcomed its first external committer. There are now more external committers in Orekit than CS-SI committers.

5. CUSTOMER USERS

The business model that CS-SI uses for Orekit is mainly a classical service-based model. CS-SI does paid development and integration for its customers, either within Orekit itself or as add-on for applications built on top of Orekit. In addition to development and integration, CS-SI also provides training sessions on the library and its dependencies, but these training sessions alone would never be sufficient to sustain the project.

When an Orekit user is a CS-SI customer and does not want some elements to be public, these developments are made in a traditional set up with limited access. However, as the Orekit project is meant to be as open as possible, a specific deal is proposed during contract negociation where the parts that are generic enough and do not involve specific know-how that must be protected are contributed back to the mainstream free Orekit library. The customer-specific parts and know-how are kept in a separate layer delivered to the customer at contract acceptance. The customers benefits are that they will not have to pay the maintainance cost for the generic parts, they are assured these parts will remain in synch with the upstream library, they will benefit freely from both bug fixes and further enhancements contributed by other users. Another argument in favor of generic features contribution is that it reduces the risk that someone else will develop and contribute a competing solution for the same topic. Such competing features generally induce additional costs for the initial customers, either whereas they still want to maintain their own out-of-tree feature or whereas they decide to abandon it and adopt the new contributed feature. The benefits of contributing generic features for the Orekit project is that new features are introduced more quickly and can benefit more users.

This process works quite well, and several projects have been realized this way [9, 10, 11].

In some cases, users prefer to develop their own versions out-of-tree and maintain everything themselves. One should however be aware that this solution, as appealing as it might appear at first, is not always safe and cost-efficient. It has been studied at length and there appear to be mostly four different cases, depending on how much one depends on the upstream open-source project [12]. At one end of the choices range, the open-source dependency is an incidental small artifact for a short term project. In this case the most cost effective solution is to maintain it out of tree. At the other end of the choices range, the open-source dependency is a critical central part for a long-term project. In this case the most cost effective solution is to hire one of the maintainers or to have one of the in-house engineers become a maintainer of the open-source project. There are also some intermediate cases between these two extremes.

6. USERS REQUESTS TOPICS

The topics addressed by users requests have slightly shifted from the early years when Orekit was still in a stabilizing phase to the current phase of an established tool.

6.1. Stabilizing phase

During the early stabilizing phase, most users requests were questions. At that time, users were discovering the libary and asking themselves and the developers whereas the library could be used in their applications and how to use it.

The first-ever question asked was about library validation. It is still a question that is asked from time to time. Another set of early questions were related to initial set up, by people starting to test the library and getting stuck somewhere. As a result, the online documentation was improved and hardly get them anymore. This is one side effect of having different users: they encounter a problem and this guide the project team to improve the product, here it was the documentation that was lacking and we did not identify it alone, because we were already used to the library since years so everything seemed natural and well explained to us despite it was not.

Then some bug reports were made, sometimes with fixes. The first bug report arrived a few months after the first release, it had the fix attached to it. Here again, this report proved the interest to have a set of different users than yourselves: the bug was related to a gravity field loader that we were not using ourselves (we used a different one), so the bug would have remained in the library for a long time before being fixed if we were the only one to use Orekit and have access to the code. The bug was identified, analyzed and fixed by someone else, and we got the fix for free.

When it was released publicly, the library was already quite features-rich and included a sufficient number of components to allow developing complete space flight dynamics applications. So the first feature requests that arrived were mostly related to the API (improving data loading, providing access to some internal data ...).

The profiles of the users were also unexpected to us. As per company history, we dealt mainly with agencies and large companies like spacecrafts manufacturers. We did expect most of potential Orekit users would belong to these entities and that this would be a small world and that we would already know most of our users. Someone even presaged us Orekit would be a failure because “there is no such thing as a space flight dynamics community”. We were wrong. There are a lot of people using space flight dynamics out there. They do not necessarily belong to either agencies or big companies, some are in small consultancy business, in start-ups or in academics. Some came from big companies we never succeeded to approach before and that came to us by themselves once Orekit was released. Many of our users were CS-SI competitors (we expected it), but they also did contribute (we did not expect it).

All these users brought something to us. Sometimes it was a simple hint that documentation was not clear in a specific place, sometimes it was a fix, sometimes it was a question about a feature that gave us some guidance about what the future development priorities should be if we wanted to match users needs.

6.2. Established tool phase

Now that the Orekit tool is more established and as documentation has improved, we get less basic questions about setting up the library for simple applications. The topics of the interactions have shifted towards expert uses of flight dynamics and more mission-oriented domains. Recent examples about expert use are the ability to propagate in osculating elements starting from mean elements, or how to handle negative altitudes in ground stations corrections models. Recent examples about mission-oriented features are visibilities of complex geographic zones from an on-board sensor, or orbit determination capabilities.

These new questions showed us that Orekit is used in various contexts for real problems solving. These users are already quite fluent with Orekit and the level of the questions show they know very well the internals of the library, proving they have been regular users for months or years.

6.3. Recent examples

A typical recent example of mission-oriented feature that was asked for by several users is the ellipsoid tessellation. In fact, no users jumped to ask explicitly for ellipsoid tessellation, but the handling of several different users requests resulted in this feature to be added.

Everything started a few years ago, as for some Earth-observing mission needs, it appears necessary to detect when a spacecraft did fly over some geographical zone. The discrete events detection feature which has been included in Orekit since the beginning (long before it was free software) was the obvious mechanism to implement this, but it required a scalar detection function that would be continuous and change sign when flying over the zone boundaries. This looks simple at first, but becomes quickly tricky as we consider the corner cases related to the spherical topology. What happens if the zone crosses the antimeridan at 180° longitude? What happens if the zone contains a pole? What happens if the zone is not path-connected, for example if it consists of several islands in an archipelago? What happens if the zone contains holes, for example if you are interested in the littoral zone around an island but not in the land mass? What happens if you combine everything, looking at several big islands, containing lakes, containing small islands, containing pools, at the antimeridian? These questions arose as starting from a specific mission-related request, with in fact only simple zones, we anticipated that once available it will grab the attention of other users that will surely need to go behind those simple zones. So from the very beginning, it was decided that the answer to this user request should not be handled by the naive implementation using a simple loop of longitude-latitude points; it would clearly not scale up to more complex use cases.

This is an important part of analyzing users requests, and is even more important for low level libraries like Orekit. Designers and developers must prepare their code to be used in very different cases. They have to think out of the box. Looking only at one specific use case, one specific user needs is a sure way to fail. This characteristic of low level libraries explain why open-source is well adapted for low layers: with a wide range of users, more extensible solutions get designed. This is quite similar to how design patterns work [13].

This first need took some times to be properly handled, as the main problem was more geometry related than space flight dynamics related. It was therefore handled at the Apache Commons Math level, in the geometry package thanks to the generic Binary Space Partitioning trees implementation [14]. As the underlying mathematical models were available, they were used in Orekit to implement the detection function as a signed distance to boundary which was made available as part of Orekit 7.0.

As the new detector was available, it quickly appeared that despite complex zones handling was already anticipated and solved, unexpected user needs flourished around it that were not addressed. We were able to handle a single moving point (the spacecraft) with respect to a complex ground zone. We had already been able for years to handle a single target point (for example a ground point) with respect to an on-board sensor field of view, for either circular or dihedral sensors shapes. Users wanted to have both, i.e. when does a complex ground zone enters or leave a complex sensor field of view. Our solution was not sufficient for this new use case.

Then, in early 2015, just after the release of Orekit 7.0, a completely independent user request was made: someone needed to generate regular tiles on a geographic zone, and then to compute visibilities on the center of each tile. This was the missing piece that gave us a hint on how to solve the problem. The request had nothing to do with sensors fields of view, but as we had just finished the geographic zones work, it combined nicely and gave us the idea of using sampling to solve the other problem. Once again, combining unrelated needs from different users solved gracefully a complex problem. Figure 1 shows an example result of this tiling process.

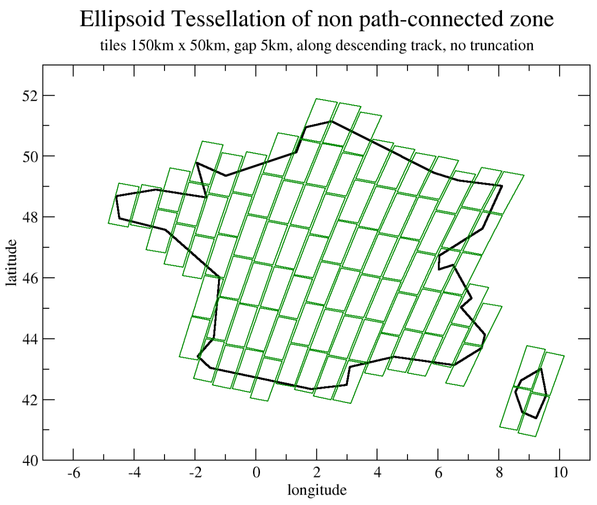

Fig. 1. Ellipsoid tessellation

Using the complex zones definition made available since Orekit 7.0, we already handled the geometrical models efficiently. So despite our user asked for simple zones, we delivered him something that could handle non-path connected zones (as in the figure with continental France and Corsica), and even more complex cases. This was already much better than existing libraries. We also took care of fine modelization of tiles directions, handling either constant azimuth with respect to the North or along track path given a reference orbit (with the maximum latitude U-turn taken properly into account), and even taking into account at each point the different radii of curvature due to the ellipsoidal shape. In just a few weeks, we were able to deliver a complex feature that exceeded expectations, just building on top of what we had made before. The tessellation feature was contributed to the open-source library, and the complete mission-specific application was delivered separately.

We were already happy with this feature and ready to use it for solving our complex geographical zone in complex field of view problem when yet another user request came in, this time concerning dilution of precision computation over a zone. Building again on what we had done, handling this request was straightforward, we just adapted the API of the new tessellation feature for also allow creating grids of points (which is simply the dual operation to creating tiles) and we were done.

Then we added the final region to field of view feature we wanted to have for years. Strangely enough, someone asked us for this just as we were developping it.

As could be expected, the story does not end here. As we were ready to finalize version 7.1, one of our external committers, noting the new region to field of view feature, noticed this could be enhanced further to be applied to ground stations, which we did not foresee. We changed slightly an API and a few days later he provided new detectors.

All these features are available in version 7.1 of Orekit, which was published in February 2016. They are the result of combining at least five different users requests in a consistent framework.

Another example shows a completely different approach. This is the orbit determination feature. This feature has been the evergreen lacking feature in Orekit. As an attempt at self-derision, when we set up the forge in 2011, we even registered this feature request as the first one in the issue tracker [15]. A small subset of users told us at the very beginning that orbit determination was a too high-level feature and that it didn't fit in Orekit scope. An overwhelming number of other users on the contrary really missed it, some of them even saying it was the only missing feature and that they were eagerly waiting for it.

As we were still thinking about it, we were happily surprised to receive a contributed orbit determination from Telespazio, despite we are more often competitors than associate [16]! This was a great success of the open-source model and we thank them a lot for this. However, it appeared merging this contribution into the library would be very difficult. It was a dedicated application instead of a library framework, its design was not in line with Orekit standards, measurements types were not object oriented, it was not extensible for users-defined measurements or qualibration factors ... So unfortunately, this contribution remained unused for several years, and we were really sorry about that.

We thought some low level building blocks should be made available in Orekit so users can easily create full-fledged orbit determination systems. We already had some of these blocks. One such block is the full integration of state transition matrices as well as partial derivatives of state with respect to model parameters in the numerical propagator (implemented using variational equations). Another block is the least squares adjustment of initial orbit used in propagator conversion. It would be nice to also provide a proper measurements interface with several Orekit-provided implementation for traditional measurement types (range, range-rate, angular, 3D point) and perhaps more exotic ones (double-range turn-around measurements, angular measurements of ground references extracted from on-board remote sensing, ...). It should be possible to include biases at different levels (for example at ground-station level where it would be different for all ground stations or at spacecraft level where it would be shared among all measurements of the same type). It would also be nice to be able to go through maneuvers and even to calibrate maneuvers. Perhaps loading CCSDS tracking data messages (CCSDS 503.0-B-1) should be included too, just as we already supported other CCSDS standards.

Of course, the goal was not to implement everything up to mission-specific loading of meta-data like special measurements formats, calibration or pre-processing features, but only to provide at Orekit level the general purpose parts and standard-compliant parts with some high-level API. It would remain user responsibility to build operational orbit determination applications from the Orekit-provided building blocks, thus remaining true to Orekit scope.

We proposed to our fellow developers to follow this approach just after the release of version 7.0, and it was well received.

However, nobody came up to tackle the problem and after a few more months this was still only a *to be done* feature. We were aware of at least four different orbit determination applications realized on top of Orekit, without being included in the library itself. Some were non-operational research tools [17], some were closed-source development made by industry or agencies. Orbit determination is indeed an important feature and designing it for reuse and extensibility is not an easy task.

So according to Orekit business model and still wanting to add it to Orekit, the feature was proposed to customers, with the deal that they will each pay for only a small part of the complete development and that the core building blocks would be contributed by CS-SI to Orekit, whereas specific top level applications built on top of these core building blocks would remain customers own. Using this model, we were able to design and develop orbit determination in the scope of Orekit, with user-extensibility in mind.

As of early 2016, the feature is complete, but still not in the main public repository. It is in one of the few non-public repositories that are used for contractual work in Orekit. It *will* be merged in a future version, expected to be published in the summer of 2016.

7. CONCLUSION

In this paper, we have traced the history of users interactions in Orekit, from the simple point-to-point discussions of the early years to the use of collaborative tools.

Orekit successfully gathered a community around the project, despite focusing on a niche domain. This community is more diverse than we expected at first, and it is helpful.

Having external users appears to be a key point for the evolution of a project, and open-source is a key to attract users. Different points of view, different priorities, different needs are strong safeguards that avoid rushing into a dead-end. These users must be cherished, they are the best asset of an open-source project.

8. REFERENCES

[1] L. Maisonobe, V. Pommier-Maurussane, Orekit: an Open-source Library for Operational Flight Dynamics Applications, in 4th *ICATT*, May 2010.

[2] Orekit site: `https://www.orekit.org/`.

[3] Orekit Apache Software License V2, `https://www.orekit.org/license.html`.

[4] L. Maisonobe, P. Parraud, S. Dinot open-source publication: a strategic choice for private companies, in 6th *ICATT*, Mar 2016.

[5] E. S. Raymond The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, ISBN 1-565-92724-9, 1999.

[6] Orekit forge: `https://www.orekit.org/forge`

[7] Orekit Governance: `https://www.orekit.org/forge/attachments/download/338/OREKIT_Governance.pdf`

[8] L. Maisonobe, A. Espesset, G. Prat, Rugged: an open-source sensor-to-terrain mapping tool, in 6th *ICATT*, Mar 2016.

[9] J.M. De Juana, S. Pessina, D. Aguilar, High Fidelity End-to-End Orbit Control Simulations at Eumetsat, in 23rd *ISSFD*, Nov 2012.

[10] N. Bernard, L. Maisonobe, L. Barbulescu, P. Bazavan, S. Scortan, P. J. Cefola, M. Casasco, K. Merz, Validating Short Periodics Contributions in a Draper Semi-Analytical Satellite Theory Implementation: the Orekit Example, in 25th *ISSFD*, Oct 2015.

[11] L. Maisonobe, J. Seyral, G. Prat, A. Espesset, Rugged: an operational, open-source solution for Sentinel-2 mapping, in *SPIE Remote Sensing 2015*, Sep 2015.

[12] Dave Neary, The Cost of Going it Alone, `https://desktopsummit.org/sites/www.desktopsummit.org/files/cost_going_alone.pdf`, Aug 2011, Accessed: 2016-02-24.

[13] Design patterns: `https://en.wikipedia.org/wiki/Software_design_pattern`

[14] Apache Commons Math: `https://commons.apache.org/math/`

[15] Orekit issue 1: `https://www.orekit.org/forge/issues/1`

[16] Telespazio contribution: `https://www.orekit.org/forge/projects/orbit-determination-telespazio-s-contribution`

[17] E. M. Ward, J. G. Warner, L. Maisonobe, Do Open Source Tools Rival Heritage Systems? A comparison of tide models in OCEAN and Orekit, in AIAA/AAS Astrodynamics Specialist Conference, SPACE Conferences and Exposition, (AIAA 2014-4429)