Design Patterns for GNSS Software Receivers

Carles Fernández-Prades*, Carlos Avilés†, Luis Esteve‡, Javier Arribas*, and Pau Closas*
*Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Castelldefels, Spain.
Email: {carles.fernandez, javier.arribas, pau.closas}@cttc.cat.
†Universitat Oberta de Catalunya (UOC), Barcelona, Spain. Email: carlos.aviles@googlemail.com.
‡Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. Email: luis@epsilon-formacion.com.

Abstract—The aim of this paper is to provide guidelines for developing high-quality, well-written GNSS software receivers. The systematic application of software design patterns and programming methodologies speeds up the development process by providing tested, proven development paradigms, and improved code readability for coders, receiver architects, and users. This includes unit testing of all the modules, and the use of a code programming style stressing readability and language-specific conventions. All these good practices result in fewer bugs, reduce reinvention-of-the-wheel, and cut long-term maintenance costs. The paper provides an analysis of design patterns, good coding practices, and methodologies for the design and implementation of GNSS software receivers, describing a working implementation released under the GNU General Public License.

I. INTRODUCTION

In Global Navigation Satellite System (GNSS) receivers, signal processing techniques dealing with the challenges posed by the multi-constellation/multi-frequency concept and the requirements of scientific applications should be designed along with the receiver architecture. The software defined radio approach is a design trend that is reaching a certain degree of maturity, moving from a strictly-research orientation to higher levels of performance, reliability, and robustness, shortening the path to meet the stringent requirements of scientific and industrial applications. In this sense, a discussion on software design patterns and their application to multi-constellation/multi-frequency software receivers appears as a timely, essential task to achieve the goals of efficiency, modularity, interoperability, and flexibility demanded by different user domains (aeronautical, land mobile, maritime, earth observers, geodesy, etc).

In this paper, the approach is based on modern computer science concepts, ranging from design patterns and template metaprogramming (which shifts complexity to compile time instead of execution time, giving place to more optimized executables) to agile software development techniques, including software development cycles consisting in planning, requirements analysis, design, coding, unit testing, and acceptance testing. All these constitute a set of engineering best practices intended to allow for rapid delivery of high-quality software.

In this context, this paper presents an open-source GNSS software receiver released under the GNU General Public License (GPL), thus ensuring the freedom of modifying, sharing, and using the code for any purpose. This secures practical usability, inspection, and continuous improvement by the research community. The proposed receiver provides an interface to different suitable radio-frequency (RF) front-ends and implements all the receiver chain up to the navigation solution. Its design allows any kind of customization, including interchangeability of signal sources, signal processing algorithms, interoperability with other systems, output formats, and offers an interface to all the intermediate signals.

The goal is to write efficient and truly reusable code, easy to read and maintain, with fewer bugs, and producing highly optimized executables in a variety of hardware platforms and operating systems.

In that sense, the challenge consists of defining a gentle balance within level of abstraction and performance. The proposed software receiver runs in a commodity PC and provides interfaces through USB and Ethernet to a variety of either commercially available or custom RF front-ends, adapting the processing to different sampling frequencies, intermediate frequencies and sample resolution. This makes possible rapid prototyping of specific receivers intended, for instance, to geodetic applications, observation of the ionospheric impact on navigation signals, signal quality monitoring, or carrier-phase based navigation techniques. It also paves the way to effective hybridization of GNSS with other technologies (e.g. wireless communication systems, or inertial measurement units), since open access and interaction with all receiver stages is provided. Testing is conducted both by the systematic functional validation of every single software block (following a test-driven developing approach and using unit testing as a verification and validation methodology), and by experimental validation of the complete receiver using synthetic signal generators with controlled parameters and real signals captured by a RF front-end.

II. STATE-OF-THE-ART

Last decade has witnessed a rapid evolution of GNSS software receivers. Since the first GPS Standard Positioning Service software receiver described in [1], where the concept of bandpass sampling (or intentional aliasing) was introduced, several works were devoted to architectural and implementation aspects. For instance, [2] provided details about analog-to-digital conversion, high sensitivity signal acquisition and different tracking loops, and [3] discussed real-time issues such as the transition from acquisition to tracking. Textbooks [4] and [5] increased the awareness of the community about the great benefits provided by software receivers with respect to the traditional hardware-oriented approach, providing a Matlab implementation of a complete GPS receiver. In order to accelerate computations and attain real-time in commodity general-purpose processors, bitwise operations were introduced in [6]. The use of Single-Input Multiple-Data (SIMD) parallel computing technology for the correlators and other time-critical operations is due to [7], a solution that exploited an extension set of assembly instructions for Intel processors. Both approaches suffered from being bit-depth dependent, jeopardizing flexibility since signal quantification cannot be easily changed. Other recent approaches take advantage of today’s pervasive multicore architecture processors [8], [9], or of the computational power of modern Graphics Processing Units (GPUs) [10]. Test procedures for GNSS software receivers were addressed in [11], and a general discussion about the architecture is found in [12]. Carrier phase measurements and clock steering are discussed in [13].

Today, there are available solutions at university and commercial levels, usually not only including programming solutions but also the realization of dedicated RF front-ends. As examples, we can mention the GNRx (GNSS Software Navigation Receiver) developed by the
Position, Location And Navigation (PLAN) Group of the University of Calgary [12], the ipexSR, a multi-frequency (GPS C/A and L2C, EGNOS and GIOVE-A E1-E5a) software receiver developed by the Institute of Geodesy and Navigation at the University FAF Munich [14], or N–Gene, a fully software receiver developed by the Istituto Superiore Mario Boella (ISMB) and Politecnico di Torino that is able to process in real–time the GPS and Galileo signals broadcast on the L1/E1 bands, as well as to demodulate the differential corrections broadcast on the same frequency by the European Geostationary Navigation Overlay Service (EGNOS) system. This receiver is able to process in real–time more than 12 channels, using a sampling frequency of approximately 17.5 MHz with 8 bits per sample [15].

Regarding observable processing and data management, the GPS Toolkit (GPSTk) [16], [17] is an open source project that provides a GNSS computing suite to the satellite navigation community, consisting of a core library, accessory libraries, and some applications. It is also worthwhile to mention the NAvigation Package for Earth Observation Satellites (NAPEOS) software [18], used by the navigation support office (OPS–GN) at the European Space Operations Center (ESOC) since January 2008 for all its International GNSS Service (IGS) activities [19].

III. PROGRAMMING METHODOLOGY

After a brief review of the state–of–the–art, this Section aims to provide concept definitions, practical guidelines, good coding practices, and development methodologies to attain the objective of well–written GNSS software receivers.

A. Design patterns

The concept of software design pattern was firstly introduced in [20], a book that rapidly became a fundamental reference on the topic. Software design patterns are descriptions of solutions to common software problems arising in different contexts, capturing recurring structures and dynamics among software participants to facilitate reuse of successful, thoughtfully proven designs. They generally codify expert knowledge of design strategies, constraints and best practices. Following a pattern helps to resolve key design forces such as flexibility, extensibility, dependability, predictability, scalability, and efficiency. They are not code recipes but generalized solutions to commonly occurring problems, showing relationships and interactions between classes or objects but without specifying the final application instantiations. The work in [21] took these concepts further, and explored their applicability to real–time systems. Other references are [22], dealing with concurrency issues, [23], that introduced the object pool pattern, and [24], that advocates using patterns for improving an existing design.

Patterns can be classified into architectural, when they apply to large–scale organization of subsystems and components, how they are constructed or how are they managed, and mechanistic, when they are more local in scope and define mechanisms for object collaboration. This latter type can be subdivided into categories of creational (patterns that deal with object creation mechanisms), structural (identifying ways to realize relationships between entities), or behavioral (identifying common communication patterns between objects that increase flexibility). Hereafter we provide descriptions of useful design patterns concerned to GNSS software receivers.

1) Architectural patterns: In GNSS receivers, architectural patterns are related to the system organization into sets of sequential transformational elements (where actual signal processing is performed), and how to handle concurrency and memory management. They solve and delineate some essential cohesive elements of the software architecture.

Name: Channel Architecture pattern [21].
Problem: Algorithms process the input data stream, applying the same set of operational transformations, such as acquiring or tracking the signal of different satellites. It is desirable an architectural structure that improves throughput capacity (i.e., the number of satellites to be tracked) with the replication of architectural units, allowing efficient parallel processing of data.
Solution: Group all the signal processing related to a single satellite into a channel subsystem. A channel can be thought of as a pipe that sequentially transforms data from an input value to an output value.
Consequences: The Channel Architecture pattern is well suited to the sequential transformation of data from one state or form to another. It simplifies algorithms that can easily be decomposed into a series of steps operating on isolate elements from a data stream. Instances of channel subsystems can be added to enlarge the number of processed satellites. The architecture is easily adaptable to handle multiple elements of the data stream in parallel, even when they are at different stages of processing.

Name: State Machine pattern [25].
Problem: It should be possible for an object to alter its behavior when its internal state changes (for instance, a lost of tracking event)
Solution: A State Machine consists of the following parts: i) an Automata interface implemented by the context that is the only way of interaction between the automata and a client, ii) a Context class that encapsulates transition logic, implements the automata interface and holds an instance of the data model and the current state, iii) State classes that determine the behavior in a particular state, each of them implementing the automata interface, iv) Events initiated by the state classes and passed to the context that does a transition depending on the event and the current state, v) an Event notification interface implemented by a context, the only way of interaction between the state classes and the context, and finally vi) a Data model, a class to provide a shared storage between the state classes.
Consequences: The State Machine pattern makes the state–dependent behavior to be localized in the state classes. The transition logic is separated from the behavior in a particular state. The state classes should only notify a context of a particular event. This pattern does not contain redundant interfaces for the context and the state classes they all implement the same interface. The implementation of an interface is trivial and could be generated automatically. Transitions could be implemented as a simple index lookup. Also note that the automata interface is implemented by the context and by the state classes. This allows making certain compile–time consistency check.

Name: Message Queueing pattern [21].
Problem: In most multithreaded systems, threads must synchronize and share information with others. Two primary things must be accomplished. First, the tasks must synchronize to permit sharing of the information, for instance concurrent access to the signal sample flow or the output of an acquisition module. Second, the information must be shared in such a way that there is no chance of corruption or race conditions (condition in which a result depends on the order of execution, but the order of execution cannot be predicted).
Solution: The Message Queueing pattern provides a simple means for threads to communicate information among one another. Each thread owns a message queue that stores messages received asynchronously. When the thread is active, it reads messages from the queue and
processes them, usually by dispatching them to an internal object. Each queue is protected by a mutual exclusion semaphore, since the queue itself is a shared resource (shared between the owning thread and the thread objects that want to send it messages), it must be protected from simultaneous access. This is a well-known approach to the producer–consumer problem that frequently arise in multi-threaded systems, particularly used to decouple processes that produce and consume data at different rates.

**Consequences:** This pattern is supported by virtually all real-time operating systems and multitasking languages. The primary disadvantages are that it is a relatively heavyweight approach to information passing among threads, it does not necessarily allow for highly efficient information sharing (since it is asynchronous, the receiving thread does not process the incoming message until it becomes ready to run), and information must be shared by value instead of by reference. This limits the complexity of the information sharing possible and does not allow for efficient use of large data structures.

**Name:** Smart Pointer pattern [21], [26].

**Problem:** When dealing with pointers, precise management is easy to forget about when dealing with all possible execution paths. Inevitably, somewhere a pointer is destroyed (or goes out of scope), but the memory is not properly freed (a memory leak), memory is released but nevertheless accessed (dangling pointer), or memory is accessed but not properly allocated (uninitialized pointer). These problems are notoriously difficult to identify using standard means of testing and peer reviews.

**Solution:** The basic solution of the Smart Pointer pattern is to reify the pointer into an object that contains the actual pointer as an attribute, as well as constructor, destructor, and access operations, thus preventing inappropriate use.

**Consequences:** The main advantage of applying this pattern is that it is a simple means to ensure that objects are destroyed when they are no longer accessible, for instance the tracking object of a satellite already lost. To ensure safety, no raw pointers should be used.

2) **Mechanistic patterns:** In GNSS receivers, we have identified creational (Factory Method), structural (Facade), and behavioral (Template Method) patterns, that are described hereafter.

**Name:** Factory Method pattern [20].

**Problem:** One of the most attractive features of a software receiver is the possibility of interchanging algorithms (for instance, different implementations of signal acquisition and tracking) and observe its impact in the whole system, or establish fair comparisons among them. The creation of such objects often requires complex processes not appropriate to include within a composing object. The object’s creation may lead to a significant duplication of code, may require information not accessible to the composing object, or may not provide a sufficient level of abstraction.

**Solution:** Encapsulate the processes involved in the creation of objects. Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses (see Figure 1).

**Consequences:** Factory Methods eliminate the need to bind application–specific classes into the code, provide hooks for subclasses (thus making more flexible the creation of objects inside a class with a factory method than creating an object directly, for instance the addition of a new tracking method), and connect parallel class hierarchies (thus localizing knowledge of which classes belong together). A potential disadvantage of factory methods is that client applications might have to subclass the creator class just to create a particular concrete–product object.

**Name:** Facade pattern [20].

**Problem:** A common design goal is to minimize the communication and dependencies between subsystems.

**Solution:** Provide a unified interface to a set of interfaces in a subsystem, defining a higher–level interface that makes the subsystem easier to use.

**Consequences:** The Facade pattern shields clients from subsystem components, thus promoting weak coupling among them. This allows changes in the components of the subsystem without affecting its clients, reduces compilation dependencies, simplifies portability to other platforms, and it does not prevent applications from using subsystem classes if they need to.

**Name:** Template Method pattern [20].

**Problem:** We would like to improve the implementation of a certain part of an algorithm without having to reprogram everything.

**Solution:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

**Consequences:** Template Methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes. It is important for template methods to specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

In summary, using patterns helps in the reuse of proven software designs and architectures, preserve crucial design information, guide design choices for application developers, and they create a common terminology, of importance when working on teams because it simplifies communication between team members. This is of particular importance in interdisciplinary groups, as is usually the case in the development of GNSS software receivers.
B. Computer programming style

Since the seminal work by Kernighan et al. in 1974 [27], there is a clear concern on the style in writing software and its impact in the final quality of the product. Following programming guidelines and code conventions not only helps to avoid introducing errors, but cuts maintenance costs and favours effective code reuse.

Generic programming is a well-established style in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. Software entities created under this paradigm are known as parameterized types [20], or templates when using C++. Templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. This mechanism, known as static polymorphism, along with being a way of pre-evaluating some of the code at compile-time rather than at run-time, also reverts in more optimized code, smaller executables, shorter runtimes, and lesser memory requirements, avoiding the overhead of run-time polymorphism. Compile-time execution refers to the ability of a compiler, that would normally compile a function to machine code and execute it at run time, to execute the function at compile time. Since in a software receiver working in real-time performance is critical, we should maximize compile-time work (which usually is not an issue), and even in an offline, not time-restricted signal processing this methodology provides benefits such as syntax and semantic analysis, code generation, timings, and earlier bug detection.

Since programming styles are often designed for a specific programming language, let us first discuss the choice of C++ for the GNSS software receiver presented in this paper. The rationale relies on the fact that C++ is a dominant language, it is much easier to recruit experienced programmers, and a number of well-written, peer-reviewed libraries are available. Besides that, C++ is truly a language for multi-paradigm development: the language supports generic, imperative, and object-oriented programming, allowing use of templates, static polymorphism, and generic programming. In the absence of instantiation bounds, C++ templates have been shown to be Turing-complete [28], which, roughly speaking, means that they are powerful enough to compute anything – any computation expressible by a computer program.

The evolution of C++ is directly linked to the evolution in the understanding of design patterns and generic programming concepts. When the C++ standardization process was started in 1989 by an international ANSI/ISO committee, it also included the development of a standard library that extended the core language to provide some general components, a set of common classes and interfaces: the standard template library (STL), special containers, strings and numeric classes, input/output streams, etc. These give programmers a higher level of abstraction [29]. Textbooks [30] and [26] provide a number of practical examples of judicious uses of the STL for producing effective and efficient code. The next step in generic programming applied to C++ is Boost [31], [32], a set of free, expertly designed, peer-reviewed portable C++ source libraries. Boost provides reference implementations that are suitable for eventual standardization. Actually, some of the Boost libraries are already included in the current C++ standard and several more are expected to be included in the new standard now being finalized. The use of STL and Boost libraries helps to produce better code and to increase productivity. However, an incorrect or indiscriminate usage could lead to code bloat, and thus careful selection is strongly encouraged.

As a necessarily non-comprehensive list of good coding practices, we can mention the need of clear functional and detailed specifications (requirements, architecture, analysis, design, and testing) before jumping into programming, the observation of naming conventions (set of rules for choosing the character sequence to be used for identifiers which denote variables, types, classes, etc.), code documentation, and the absence of magic numbers (unnamed or ill-documented numerical constant values). References [30], [33], [34] provide a wide-scope list of good coding practices.

C. Software development methodology

This subsection defines the framework that is used to structure, plan, and control the process of developing a GNSS software receiver.

1) Test-driven development: We propose a test-driven development approach [35], [36], consisting in repeating test-case cycles (sets of conditions or variables under which a tester will determine whether an application or software system is working correctly or not) instead of a feature-driven approach. This methodology is claimed to offer valuable benefits to software development: facilitates change, simplifies integration, automatizes documentation, helps to separate the interface from the implementation, increases developers productivity, and plays a central role in the software quality assurance process. In this methodology, the tests are written before the functionality that is being tested. This has shown two main benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. There are different testing levels: unit testing (that checks the functionality of a specific section of code), integration testing (checks the interfaces between components against a software design), system testing (tests a completely integrated system to verify that it meets its requirements), system integration testing (verifies that a system is integrated to any external or third party systems defined in the system requirements), regression testing (focuses on finding defects after a major code change has occurred), and user acceptance testing.

- **Unit testing.** Understanding unit as the smallest testable part of an application, unit testing [37] checks the functionality of source code portions. It consists of three phases: i) perform the test planning (plan the general approach, resources, and schedule; determine features to be tested; refine the general plan), ii) acquire the test set (design the set of tests; implement the refined plan; and design), and iii) measure the test unit (execute the test procedures; check for termination; and evaluate the test effort and unit). Unit testing infrastructure should be lightweight and allow introspection (a language feature that allows the state of an object to be observed and altered by means of a uniform mechanism that is equally applicable to objects of all classes).
- **Integration testing.** Phase in software testing in which individual software modules are combined and tested as a group, after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.
- **System testing.** Test should also be conducted on a complete, integrated system to evaluate the compliance level with its specified requirements. These tests should require no knowledge of the inner design of the code or logic, and include performance testing [38] using profiling and logging tools (which is of relevant importance in applications targeted to work in real-time).
• Higher–level testing. User testing feedback and peer–reviewing are powerful tools in producing high–quality software. Releasing the source code under the GPL allows thoughtful inspection not only of software features but also of how they are implemented.

2) Development ecosystem. Infrastructure for project management and code development is of equal importance than programming style. A set of efficient and easy–to–use tools helps managers and developers to keep tight to objectives and schedule, avoiding improper communication and facilitating feedback among users and developers.

• Project management tool. This includes scheduling, resource allocation, collaboration software, communication, quality management and documentation or administration systems. When web–based, it should be accessed from any type of computer, ease of access–control, and multi–user. Storage “in the cloud” of relevant documents is suggested.

• Version control system. An application that automates the process of keeping an annotated history of the project, allowing reversion of code changes, change tracking, and bug tracking is essential. As examples we can mention Subversion[49] and git [40]. Trac [41] and redmine [42] are web–based tools that integrates well.

• Integrated Development Environment (IDE). Admitting that each developer has his/her own preference, a full–featured IDE increases productivity. Usually, an IDE consists of a source code editor, an interface to a compiler and/or an interpreter, build automation tools, and a debugger, but modern ones also provide integration with the version control system, a class browser, an object inspector, and a class hierarchy diagram. Eclipse[43] is a free and open source option.

• Build tool. The build process should be easily maintained and highly portable. When not thought out well, development time shifts towards build system tweaking instead of source file coding. There are several tools that automatize the process, such as bjam[44] and CMake[45]. Regarding the compiler, we suggest the GNU Compiler Collection (gcc)[46], a multi–platform, world–class optimizing compiler that closely follows language standards.

• Communication. An email distribution list uses to be an efficient communication tool among developer team members. When needed, can be complemented with net–meetings and personal instant messaging. In these cases, emailed minutes of meeting are useful to keep logging of discussions and decisions.

IV. IMPLEMENTATION

Hereafter we present a software project, so–called GNSS–SDR and accessible from http://cms.cetc.es/trac/gnss-sdr/wiki, that tries to implement the concepts presented so far. The implementation is heavily based on GNU Radio[47], a well–established framework that provides the signal processing runtime and processing blocks to implement software radio applications. A framework is a product, while a design pattern is a specification. Frameworks are a special case of software libraries – they are reusable abstractions of code wrapped in a well–defined Application Programming Interface (API), yet they contain some key distinguishing features that separate them from normal libraries: the overall program’s flow of control is not dictated by the caller, but by the framework; and it can be extended by the user usually by selective overriding or specialized by user code providing specific functionality. Software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low–level details of providing a working system, thereby reducing overall development time.

In case of GNU Radio, it incorporates many of the design patterns described in Section III–A. From an architectural point of view, a GNSS–SDR application can be seen as a factory with different working lines where we put raw data at the input and we get processed data at the output. This raw data is, in our case, the continuous stream of signal samples and the processed data is the position of the receiver updated constantly and regularly. This simplification describes quite effectively the GNSS–SDR architecture. Since the designer should focus in solving the problem of introducing the raw data into the application, distributing it through the factory line, and gathering the results at the other end, the Channel Architecture pattern provides a suitable approach for our needs. Adopting this approach has several benefits, namely that the architecture can be easily mapped with a hardware implementation of any signal processing device, it implies a degree of modularity that is required by software defined radio (SDR) applications, the concepts involved do not require deep knowledge of software engineering concepts and can be used in an interdisciplinary environment, and the fact that using a design pattern introduces a component of standardization that is always desired for software design. GNU Radio incorporates a scheduler that assigns working threads to each block, hiding all the complexity behind a simple and robust API, uses shared memory to manage efficiently the flow of data between blocks, and offers a large set of well–programmed blocks that provide implementations for very common signal processing tasks. In contrast, GNU Radio does not provide any standard way to provide control over the blocks. Once they are connected, the application can run and data will be put into the stream. As long as there is data, the working threads will run the code of the different blocks.

A. Control

In practice, the composition of the received GNSS signals will change over time. Initially, some satellites will be visible and their signals will be received by the front–end. GNSS–SDR will need as many channels working in parallel as visible satellites there are. After a while, some satellites will not be visible anymore and new ones will show up. Some channels will lose track of their signals and some new channels will have to be instantiated to process the new signals. This means that the receiver must be able to activate and deactivate the channels dynamically, and it also needs to detect these changes during run–time. We used the Message Queuing pattern to implement a control architecture to the application. This approach is very flexible and can be adapted to almost any control we need to implement. There is a control thread that runs in parallel to the flowgraph, receiving notifications that trigger changes in the application. Some of these notifications will be sent directly from the processing blocks. For instance, an acquisition block that finishes its processing and detects a satellite’s signal, will send a notification to the control thread indicating its success. The control thread will then change the internal configuration of the channel and pass the estimations to the tracking blocks. Since GNSS–SDR is a multithreaded application, the control thread will receive messages form many sources and probably at the same time. The mechanism for sending and reading these messages must be thread–safe. The control thread has a thread–safe queue and it shares an instance of this queue with all other modules that require sending messages to

1Although we provide web references, we suggest readers interested in the forthcoming mentioned software tools, denoted in sans–serif, to look for the most updated information in the web search engine of their choice.
GNSS software receivers, aiming to discuss some formal aspects, the lack of a comprehensive quantitative measure of software quality and help the developers to focus their optimization efforts.

Automatize the profiling process, identify computational bottlenecks, and correlate the data. Relevant parameters of RF front-ends are the frequency band, the sampling frequency, the intermediate frequency, and the number of bits per sample. The signal source could be also a file stored in a hard disk.

**B. Configuration**

Configuration is a key feature in SDR applications. We need to define what signal source we want to use, and what are the characteristics of the channels (system, band, certain acquisition or tracking algorithm,...), and to specify parameters to all these modules. And we want all this to be easy and intuitive. Since it is difficult to foresee what future block implementations we will need in terms of configuration, we used a very simple approach that can be extended without a major impact in the code. This can be achieved by simply mapping the names of the variables in the blocks with the names of the parameters in the configuration. The application will only define a simple accessor class to fetch the configuration pairs of values and pass them to a factory class. This factory will decide, according to the configuration, which class needs to be instantiated and which parameters should be passed to the constructor. With that approach, adding a new block that requires new parameters will be as simple as adding the block class and modifying the factory to be able to instantiate it. This loose coupling between the blocks implementations and the syntax of the configuration enables extending the application capacities in a high degree. It also allows to produce fully customized receivers, for instance a benchmark for acquisition algorithms, and to place observers at any point of the receiver chain, which is very useful for debugging, educational, and research purposes.

**C. Algorithms**

The implementation has to resolve design forces that sometimes can be antithetical, such as flexibility vs robustness, or portability vs efficiency. With the objective of attaining real–time in mind, efficiency should be addressed specially in those blocks that works with high data rates (mainly, signal conditioning, Doppler removal, and correlation), while other blocks working at medium rate (tracking, extraction of navigation parameters) or low rate (measurement generation, navigation solution) can be implemented targeting robustness and reliability.

In the implementation of digital signal processing algorithms, the first step is an obviousness sometimes forgotten: go and find the standards. There are reference documents available for GPS L1 C/A [51], the forthcoming civilian–use signal called GPS L1C [52], GPS L2C [51], GPS L5 [53], the standard precision service of GLONASS [54], Galileo’s Open Service [55], the Japanese Quasi-Zenith Satellite System (QZSS) [56], and the definition of the message format for Satellite–Based Augmentation Systems (SBAS) such as WAAS [57, Appendix A] or EGNOS [57, 58].

Regarding the signal source, we suggest to implement a driver compliant with the VITA 49 Radio Transport (VRT) protocol [59], an emerging standard for SDR applications that was developed to provide interoperability between a diversity of SDR components by providing a protocol to convey digitized signal data and sensor settings. These would allow an easy accommodation of signals coming from a multiband RF front-end, or other sources of information providing data at different rate, such as Inertial Measurement Units for GNSS/INS hybridization purposes.

After the signal source (see Figure 2), a signal conditioner is in charge of resampling the signal and deliver a reference sample rate to the downstream processing blocks, acting as a Facade between the signal source and the synchronization channels. These channels can be managed according to the State Machine pattern. As suggested in [60], channels can be in one of the following states: idle, acquisition, confirmation, pull–in, and tracking. Satellite states can be usable and unusable, according to the knowledge and health of the almanac and ephemeris data.

Both acquisition and code/phase tracking processes are active fields of research, and a primary design goal of GNSS–SDR is to allow a rapid implementation of novel approaches and a smooth embedding in a complete receiver, as well as allowing fair comparisons and its impact assessment in the overall performance. The level of abstraction provided by the Factory Method pattern is intended to hide the low–level integration work to the programmer, and thus focusing the efforts in algorithm implementations instead of in the interaction with the rest of the receiver.

Software libraries for signal processing have to be carefully selected. We have found useful uBLAS [61], a Boost’s C++ template class library that provides basic linear algebra operations such as vector and matrix multiplication, and FFTW [62], an efficient implementation of the Fast Fourier Transform commonly used in the acquisition process.

The output of the tracking and navigation data modules can be dumped into a file (we suggest to follow the Receiver Independent Exchange Format RINEX Version 3.01 [63], which allows the accommodation of observation, navigation, and meteorological data for GPS and its modernization plans, GLONASS, and Galileo, and offers a direct interface to high–precision navigation libraries such as GPSTK) or directly delivered to another module in charge of computing the navigation solution, or to another application.

For unit testing, we have found the Google C++ Testing Framework (gtest) [64] useful and lightweight. Other used tools are the Google Logging Library (glog) [65] for application–level logging, Google Commandline Flags (gflags) for command line flags processing, and Google Performance Tools (google–perfutil) [66], a collection of a high–performance multi–threaded memory allocators implementation, plus some pretty nifty performance analysis tools that allow to automatize the profiling process, identify computational bottlenecks, and help the developers to focus their optimization efforts.

**V. Conclusions**

Design patterns encapsulate experience and knowledge, alleviating the lack of a comprehensive quantitative measure of software quality and formalizing good practices in software development. In this paper, these concepts have been applied to the specific field of GNSS software receivers, aiming to discuss some formal aspects,
associated design, and development methodologies. Besides, this paper also presents an open source GNSS software receiver that partially implements the proposed approach and offers a free-access framework for further development.

ACKNOWLEDGMENTS

Authors are especially grateful to Christian Pomar i Berry for his wise recommendations, inspiring advices, and many valuable discussions in those issues related to Computer Science and high-performance programming. This work has been partially supported by the Spanish Science and Technology Commission: CENIT2007-2002 (TIMI) and TEC2008-02685/TEC (NARRA), and by the European Commission in the framework of the FP7 Network of Excellence in Wireless COmmunications NEWCOM++ (contract n. 216715) and COST Action IC0803 (RFCSET).

REFERENCES


[50] SiGe Semiconductor, “SEA120L GNSS receiver ic datasheet;” May 26, 2009, Ottawa, ON, Canada.


