Aircraft Simulator Designing based on Object Oriented Methodologies

Rahul Kosarwal
M.E. Student, Department of Information Technology
Medicaps Institute of Technology & Management
Indore (M.P), India
rahulkosarwal@gmail.com

Anshul Kosarwal
Assistant Professor, Department of Computer Science
Acropolis Institute of Technology and Research
Indore (M.P.), India
anshul.kosarwal@gmail.com

Abstract

Object plays a very important role in comparing software entities. But the role of object when it comes to simulation field is still undefined. Several standards have been created with the purpose to provide a design solution for the project of Simulators. The design solution reported in this paper aggregates the principles of both software and simulator architectures. The objective is to invert the top-down strategy of model-driven development with the Simulation Model Portability (SMP) standard into a bottom-up development process of a SMP framework. The paper also provides solution for two development lines of two different frameworks: the first is the SMP framework that changes according to the design models, and the second is a framework designed to support the development of reusable behavior implementations.

Keywords- simulation mode portability (SMP); simulators; object oriented methodology; man-Machine Interface; model-Driven Architecture

1. Introduction

In object orientation, the distinctive features of an object are expressed in interfaces which are used to specify the interactions between objects. In these conditions, the planning of the coding activity can be done by describing the objects and the business logic they share. If these descriptions are written before any programming language specialization, we can call them design artifacts or design models. A different approach to the design of software is the specification of code functionalities through test code. Using only test code it is possible to design a piece of software before entering the stage of source code development. Similarly to design models, test code is a design artifact, but it does not follow a predefined semantic scheme as per design models. A different approach to the design of software is the specification of code functionalities through test code. Using only test code it is possible to design a piece of software before entering the stage of source code development. Similarly to design models, test code is a design artifact, but it does not follow a predefined semantic scheme as per design models. Nonetheless, test-driven development is a restrained process, carried out in closed loop through source code refactoring. While the model-driven design approach is appropriated for the development of component-based software systems, the technique of test code is well suited for the development of software frameworks.

Rather than refining the code headers derived from the design models, the behavior implementations are developed separately and without a direct knowledge of the SMP2 framework. Two phases of decoupling are foreseen: the first decouples interface descriptions from code headers, and the second decouples code headers from behavior implementations. The transformation between design models and code headers is automatized by the model-driven development environment, but the linking to the behavior implementations must be made manually. The code that effectuates this task is designated as mission specific. The articulation of both framework and their components is accomplished by the GNU Build System. This tool provides modeling languages to specify dependencies between source packages and automate the process of make le generation. Together with mission specific code, the possibility to model software dependencies at code level gives the developers the chance to combine components of both frameworks easily.

In our present time, the software systems that we project can be of great complexity in spite of the fact that computer power is continually increasing. Software maintainability is one of the main problems that software engineers are facing nowadays, which consists on the difficulty to keep their computer programs correctly written and understandable, at the same time as they expand in the number of functionalities provided. The scenario of software crisis is something which is originated when the complexity and the expectations of a software system not only increases to a great extent, but also changes very quickly. As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

The natural consequence of this present tendency are the difficulty in developing software without bugs and keeping it well documented. For great difficulties, great solutions are required. Modern software requires renewed processes of development, whereas we consider the design or the programming techniques. The persistence and development methodologies that I had to apply in the
The benefit of having an assembly design phase, which is essential in component model based systems, is the possibility to define an assembly configuration with a variable number of Model instances. This feature is called dynamic assemblage. For that purpose, the SMP2 standard specifies that all the necessary methods to navigate through the SMP2 Models' hierarchy are generic and automatically generated from the SMP2 Catalogue files. As an example for the Component-based design approach, let us consider that in a SMP2 Catalogue file is defined the Model A that the Model B has as child. Whatever is the number of Model B instantiations defined in the SMP2 Assembly file, the *GetChildren ()* generic method will return every instances of B. It is in this sense that the modeling is dynamic. It is not necessary to change the SMP2 Catalogue file in order to reconfigure the runtime component assemblage specified in the SMP2 Assembly file. In resume, the SMP2 Models are modeled by conceptualizing generic entities that act on the behalf of interface providers or interface consumers, event sources or event sinks, etc., those which can be connected together; any SMP2 Model which provides an interface can be connected to any model which has an interface link to that model; the navigation throughout the list of interface consumers is made using generic methods, thus enabling dynamic assemblage. Whereas the definition of SMP2 Models is made within SMP2 Catalogue files, the connections between the Models are specified in SMP2 Assembly files. The positive thing about this two-stage modeling is that it is possible to have more than one runtime configuration for the same SMP2 Catalogue.

Figure 1. illustrates what could be an assemblage of SMP2 Models, described in a SMP2 Assembly file.

Each design pattern is described by a different symbol and any numbers of instances are accepted. The interfaces on top of each SMP2 Model are the SMP2 generic interfaces that provide the access to the runtime platform services and to the SMP2 Model instances in which each SMP2 Model depend.

The SMP2 Models described in SMP2 Catalogue files are foreseen to be software components which are able to communicate between them in a transparent way, that is, by the means that the SMP2 component model itself provides. However, the SMP2 component model is not the traditional kind of component model used to design distributed computing systems in which the location transparency is the most important feature. Indeed, the resulting assemblage of a hierarchy of SMP2 Models may consist in a simple set of shared dynamic libraries installed in a single operation system machine. Moreover, the focus of the SMP2 component model is to give transparency in terms of method invocation on each SMP2 Model, that is, access transparency, and relocation transparency which, in this context, means that an instance of a SMP2 Model can be replaced by another instance without being noticed by the SMP2 Models which depend on it.

The dynamic configuration feature of the SMP2 component model is provided by a set of generic interfaces that all SMP2 Models know and implement so that, from the
root SMP2 Model, all the SMP2 Models can be reached by navigating throughout the root SMP2 Model's hierarchy. However, it is up to each SMP2 Model to cast the instances supplied via the generic access interfaces in order to obtain the appropriated object pointers. The same thing happens at code level where programmers need to have the C++ header of a given class in order to compile their programs and call methods on that class. Objects are declared with public interfaces which constitute the contract that each object share with the rest. At the most, what the SMP2 component model provides is additional interfaces for generic and unified communication between software components, and that is precisely where the interface overhead resides. SMP2 interfaces can be generic or Model specific, but only the generic interfaces are strongly-coupled. Nevertheless, the headers containing all the interfaces must be known at compile time. The modeling of SMP2 Models as software components must take into consideration this important statement: too much refined modeling can lead to an excessive overhead. In resume, the SMP2 component model offers some flexibility in terms of dynamic runtime access to SMP2 Models, but each SMP2 Model must know a priori, that is, in compile time, which are the interfaces provided by the Models it relates to.

3. State of Art
It starts with an inspection of the SMP2 standard, in special, of the SMP2 component model, and introduces the important feature of dynamic configuration of SMP2 components. Then, follows an insight over the Model-Driven Architecture methodology that places the SMP2 standard in the context of a model-driven environment. The gap inherent to the model-driven prototyping, that is, the development of code, is the motivation for the adoption of Test-Driven Development methodologies, which systematic processes can be very helpful for specifying and developing quality software. Another field of study with strong relevance in my work is the subject of software frameworks. The concepts of software refactoring and design patterns are presented in detail as they are crucial elements in many aspects constructing a framework. Finally, is given a glimpse of the GNU Build System. The objective of applying this technology to the development of Aircraft Simulators is the inclusion of modeling feature at code level in the model-driven environment. A deep understanding of these five topics is essential because they interconnected in the prototype development processes.

3.1 The Simulation Model Portability 2 Standard
The purpose of the SMP2 standard is to promote portability of models among different simulation environments and operating systems and to promote the reuse of simulation models. The Simulation Model Definition Language (SMDL), which is one element of the SMP2 standard, is based on the paradigm of Object Orientation (OO) and its semantics defines several design patterns, such as Interface based, Component based, Event based, and Data-flow based. In addition to these design options, the standard promotes the storage and integration of design artifacts in the XML format, namely in XML files compliant with (SMDL) XML Schema. The support of persistence in XML is the key that enables the decoupling of the design phase from the implementation phase, and the consequent possibility to reuse design artifacts.

The figure illustrates the two SMP2 Models, each one having a SMP2 generic interface, and two pairs of possible connections between the Models. The facet-receptacle pair is used when the Interface-based design pattern is applied, whereas the pair source-sink for events is used for Event-based modeling. Supposing that the design solution for modeling the association between the two Model is Interface-based, it is necessary to connect the provider facet of the Processor Model to the receiver receptacle of the OnboardSoftware Model. The SMP2 entities and their associations are described in SMP2 Catalogue files, but in order to instantiate the SMP2 Models and specify how the links between connections are effectuated, different XML documents are used. These files are called SMP2 Assembly files and are also specified in the SMDL XML Schema.

4. Problem Statement
Two different software frameworks are defined: the SMP2 Modeling Framework and the Infrastructure Framework; each framework is constructed using different development methodologies, Model-Driven Development and Test-driven Development respectively. The link between the two frameworks is accomplished by a specific type of code called mission specific that enables the hybrid platform. It will be also made an assessment of the SMP2 standard and an analysis of the implications that the hybrid solution may have in the project's life cycle of the Aircraft Simulators.

4.1 Research Proposition
The objective of my work was is the tracing of specification guidelines of a highly-configurable software architecture for Aircraft Simulators in the Linux platform. By flexibility in the configuration of a software solution, I mean the possibility to define, at compile-time, a combination of many
software modules, with different but compatible types of interfaces, in order to build a simulation with the SMP2 component model. It was already mentioned that the SMP2 standard, more concretely the specification of SMP2 Assembly files, provide a mechanism for dynamic assemblage of SMP2 Models, the component elements of the SMP2 component model. This means that for a given hierarchy of SMP2 Models it is possible to instantiate how many instances of each SMP2 Model the designer wants. Nonetheless, each instance of a SMP2 Model has exactly the same implementation because they are instances of the very same Model previously designed within a SMP2 Catalogue file.

**Figure 3. Deconstruction of the Model-Centric Design Approach**

My proposal consists in the adoption of an heterogeneous software architecture, mixing the modeling approaches of code-only and model-only, which would allow to implement, in ANSI C++, different versions of the same design configuration of a Model made with the SMDL meta-language. The mixture of two different design methodologies is opposed to the monolithic approach of a pure model-driven software system.

**Figure 4. The Complete Automation of the SMP2 Framework**

Rather than redefining the code skeletons derived from the design models, the behavior implementations are developed separately and without a direct knowledge of the SMP2 Modeling framework. Two phases of decoupling are foreseen: the first decouples interface descriptions from code skeletons, which are simply headers, and the second decouples code headers from behavior implementations. The transformation between design models and code headers is automatized by the model driven development environment, but the linking to the behavior implementations must be made manually through mission specific code.

5. **Proposed Solution**

Although the object of work is the study of development methodologies, it is important to study the impact that new technologies have on the Software Requirements Specification document. Indeed, the SRS document includes technologic constraints that dictate which technologies and methodologies should be used in the development of a given set of requirements. Since the two distinct development methodologies of MDD and TDD are considered, the SRS document must specify which requirements will be developed with the SMP2 standard and those which will be developed through test code. In the sections of this chapter are given details about the way both design techniques are applied together with the GNU Build System. It is also included a set of directives for an efficient specification of software requirements, which should be capable to explore the benefits of each design technique.

5.1 **Three Levels Of Source Code**

The fact that each mission has an Aircraft Simulator is one of the reasons for enduring a process of refactoring and evolution of the Infrastructure Framework. Across missions, the source code which is mission specific is gradually migrated to the framework or to the SMP2 Catalogue files. Mission specific code is tied up with the SMP2 Model interfaces and, at the same time, bootstraps the infrastructure code. It is in this sense that I defined mission specific code as an hybrid type of code. Such code is made available to the other SMP2 Models by the means of SMP2 interfacing, but it also uses low-level infrastructure libraries which are not defined in terms of SMP2 interfaces, although these also have generic interfaces. Besides the SMP2 wrapper code, which is automatically generated from SMP2 Catalogue files, the development of code is divided between mission specific code and infrastructure generic code. These two development lines have as their driving forces two different starting points: the high-level design of SMP2 and implementation of the Infrastructure Framework. In the one hand, we have the SMP2 modeling breakdown that concerns mainly with the business logic structure of the Aircraft Simulator in terms of its parts, or components. On the other hand, we find a process of code refactoring that searches for small parts of implemented behavior which interfaces can be easily reused. The design patterns of the SMP2 standard (Interface based, Component based, Event based, and Data-flow based) define the upper micro-architecture, whereas the GOF design patterns (Adapter, Factory, Visitor, etc.) define the lower micro-architecture.
5.2 SMP2 Modeling

The terminology "SMP2 modeling" is being used to refer to the high-level design of the aircraft that mainly tries to circumscribe the logical structure of a aircraft. This logic is strongly decoupled from the behavioral aspects. For the sake of the good organization of the software, the SMP2 Models' source code is divided in skeletons, that are generated automatically the XSIM workspace, and mission specific source code. The intention in this approach of designing a Aircraft Simulator is not the reuse of code, but rather, reuse of interfaces. The logical decomposition of the aircraft is accomplished by elaborating SMP2 Catalogue files. Using the SMP2 symbology, these catalogues aggregate the puzzle pieces that the Model Designer wishes to dispose. As already mentioned, a concrete simulation configuration is specified in a SMP2 Assembly file. In this file, are defined concrete model instantiations and connections between them. Next, it is presented a summary of the software requirements specification for developing SMP2 Models on the basis of GNU source packages.

The ANSI C++ skeletons of a SMP2 Model are automatically translated from the entities declared in SMP2 Catalogue files. This type of code, which can be simply C++ headers, is packed and distributed in separated GNU source packages. The body of each SMP2 Model, this is, the mission specific code is also distributed in separated source packages. The output format of those source packages have to be dynamic shared libraries because they are used by the component model. The static library that comes out from the skeletons source package is linked with the mission specific source package, to accomplish the implementation of the abstract classes defined in the C++ headers. If the behavioral implementation that must be provided by the SMP2 Models can be found in the Infrastructure layer, then that infrastructure libraries should be reused. Thereby, the design is absolutely decoupled from code development.

The logical decomposition of the parts of the SMP2 component model should be tackled using a top-down approach taking as reference the Platform User Manual (PUM) of the aircraft. This process has a strong cognitive character because it involves thinking in abstract representations of software components that, put all together, would stand for the ideal picture of a Aircraft Simulator. In very generic terms, my proposal includes a set of heuristic rules to uniform the modeling breakdown process. A real aircraft is made of programmable hardware and associated software, dedicated hardware and data links that provide communication between the different types of equipment. Thereby:

1. Every programmable hardware component can be represented by a SMP2 logical unit, as well as the software associated.
2. Every piece of hardware providing a specific functionality can be represented by a SMP2 logical unit.
3. The messages exchanged between SMP2 logical units across data links can be modeled by an adapter logical unit responsible to mediate the communication between the interested parts.

5.3 Infrastructure Generic Libraries

The Infrastructure layer is where the reusable implementation of the Aircraft Simulator behavior resides. The principle characteristic of the infrastructure libraries is the generalization of their interfaces according to taxonomy of design patterns. The generalization of the libraries' interfaces is an effort of abstraction that looks for very refined interface definitions and then categorizes them as proven solutions for recurrent problems. The use of design patterns to standardize a library interface is a good help for the clients of that library, but also for the developers on the provider side, because it makes possible the reuse of the library without additional, application specific, business logic. The infrastructure libraries are extensions of the Infrastructure Framework. It is an important aspect that the infrastructure libraries have only generalized dependencies between them, because they have to use the communication patterns defined inside the framework.

The Infrastructure Framework has the benefit to define a set of common classes and interfaces to support the development of infrastructure libraries. The generic classes, which can have default implementations, are provided by the framework and used by the means of inheritance mechanisms used to build up the infrastructure libraries. Similarly, the framework interfaces can be implemented by the framework components, that is, the infrastructure libraries. The primary goal of the Infrastructure Framework
is to leverage the communication between infrastructure libraries and between the mission specific source code and the infrastructure libraries.

6. Conclusion

6.1 The Hybrid Design Solution Revisited

Besides, many development tools are available in the market that can bring a great deal of automation to the process of developing software. The Model-Driven Architecture standard is one of the most successful methodologies used in for the modeling and implementation of a software system. However, in reality, MDA is much more concentrated in the modeling phase than in the implementation phase. The profit of using MDA consists in the fact that part of the code is automatically generated from the models, but it is necessary to consider also the effort that is needed to implement all the missing source code, which is not derived from the models. If the process of developing source code is not structured and controlled by some well-defined methodology, problems like the existence of bugs, will never cease to a possibility. This is why my work foresees the use of Test-Driven Development. Having the premise that the test code is written before the source code, we can be positively sure that all the source code is free from coding defects in respect to the test specification. Nonetheless, what initially can be made simpler by dividing the parts of the software project which are developed through MDA from the ones developed through TDD, can latter become the harsh work of joining the results of both development lines. In that tough scenario, the GNU Build System is the enabling technology that is capable of modeling how the disparate units of software can be compiled and linked together. With the possibility to express the details of makefiles in a declarative language, the GNU Build System is another powerful tool which can be used in the quest for increasing levels of automation in development of software.

6.2 The Project's Life Cycle

The traditional development phases of the project of a software system start with the user requirements elicitation and specification, followed by the analysis, design and implementation phases, until the final phase of testing and validation. It is a natural consequence of this procedure that a imprecision in the requirements specification can possibly be translated in a miswritten functionality on the source code. The reason for this is that it is almost impossible to specify everything in the beginning of project, otherwise the requirements specification document would be endless. What TDD offers us is, on a very earlier stage, the alternative to specify what functionalities will be written. If we feel that with MDA is impossible to perform a precise detailed design of each logical part of the software, then we can rely on test code which is able to specify clearly the internal behavior of each logical part. Therefore, the phase of testing would not be the last of the project phases but rather synchronous with the user requirements elicitation and specification phase. Indeed, the testing phase would be inserted back into the initial specification phase. This option would give much more confidence in respect to the quality of source code that is implemented and to the conformity of that code with the initial specifications.

6.3 An Assessment of the SMP2 Standard

The SMP2 standard is not a one size fits all solution because there are parts of the software that cannot be specified in terms of SMP2. Even with the adoption of the SMP2 standard, enormous amounts of source code are produced by the outsourcing software houses without any refined control. The challenge was to conciliate the specification of the general aspects of the structure of a Aircraft Simulator, which is made in terms of the SMP2 standard, with the specification of concrete functionalities written directly in test code. Details about how software is compiled, linked and installed in the operating system were becoming clearer to me, and I found out that it is possible to integrate a vast assembly of different source code projects in order to build a particular application. Moreover, I found that there are standard ways of doing this kind of job in Linux. In reality, my proposal consists merely in dividing the development of a Aircraft Simulator in two development lines and then integrate them using the Linux standard build tools.

6.4 Resume

In conclusion, the object of my work is focused on the search path of modern methodologies and techniques for developing software using the object-orientation paradigm. The personal experience that I have in developing software using well defined methodologies and the continual interest for that perfect technical solution are the reasons that impelled me in this search path. It is a natural tendency of modern software systems, like the Aircraft Simulators, to increase in their inner complexity and to expand the range of functionaries provided to the users. The project of a Aircraft Simulator is something that can be done only by software houses extremely competent and using the most powerful technologies. With the modest contribute of my work, I tried to bring something new to the group of people that I was involved with. I worked with effort and loads of creativity to put my ideas simple and clear for the benefit of the group, but also with the objective in mind to write this dissertation where I think I was able to report with fidelity all steps that leaded me in the formalization of the final proposal for the specification and implementation of Aircraft Simulators.

7. References


