Using SysML to Partition Software Applications and Reduce Project Risk

David Hetherington
President
Asatte Press, Inc
Austin, Texas, USA
david_hetherington@ieee.org

***Abstract*— In many systems that seem to be hardware-intensive, the actual critical project risk is the software, not the hardware. In these systems, the size of the development team assigned to an individual software component is a significant direct and indirect risk. Large monolithic applications are hard to architect, harder to size, and even harder to manage. Unfortunately, classical object oriented design and UML modeling approaches have not done much to make it easier to successfully complete large software application development efforts.**

Fortunately, the emerging study of System-of-Systems engineering affords an opportunity to explore an alternative approach to assembling large software applications and that is to partition them into smaller, more loosely coupled, autonomous systems. This paper will examine the use of SysML and other emerging best practices from the System-of-Systems field to decompose software applications into autonomous pieces that can be deployed on separate pieces of hardware and developed by separate teams and even separate companies.

SysML (rather than UML) is very helpful as a starting point in designing this sort of complex hardware/software System-of-Systems. The main benefits of using SysML rather than UML come from three main improvements introduced in SysML:

1. Increased Focus on System Partitioning – The original definitions of UML did not provide strong support for system partitioning. Authors of key early textbooks included comments like: "All bets are off if distributed processing is involved"

2. Improved Interface Support – The SysML definition and community of practice has put a lot of effort into interfaces. While UML did have interfaces, the same early UML textbooks contained admonitions like: "Don't pay too much attention to interfaces because they are going to change anyway"

3. Requirements Traceability – This function was simply missing from UML.

When it comes to reducing software risk, partitioning, risk compartmentalization, and project governance are key. SysML introduces strong new support for complex software project governance that can help reduce the overall project risk. (*Abstract*)

***Keywords*— Software project risk ;SysML; UML; model-driven software engineering; model-based systems engineering; MBSE; MDSE (*key words*)**

I. Introduction

Designing large, monolithic software applications is hard. Estimating the effort required to create a new, large, monolithic software application is notoriously difficult. Once the project is started, recovering from delays can be difficult or impossible.

A. The Problems of Large Software Projects

Brooks [1] set benchmark for discussions of these challenges with his description of the development of the OS360 in which wheelbarrows were needed to deliver the project newsletters. The book continues to describe the "Management Audit of the Babel Project" as a metaphor for the communication problems experienced by large software development teams.

Most recent work such as a key paper by El Emam, and Günes [2] focuses on surveys of larger numbers of projects of various sizes and not on the particular problems of very large projects. However, the top three reasons for project cancellation identified in [2] are:

- Senior management not sufficiently involved
- Too many requirements and scope changes
- Lack of necessary management skills

Which are all indications of a large project, a large organization, or both.

B. The Advantage of Small Teams

Smaller teams are inherently more agile and productive. In the field of Systems Engineering Jenney et al. [2] describe the development of the innovative Cord 810 automobile in 1935.

Figure 1 - 1936 Cord 810 [4]

The design included numerous innovations such as hidden door hinges, disappearing headlights and others. Surprisingly, chief engineer Gordon M. Buehrig and his team completed the entire project in only a few months – a dream for the current auto industry that needs 4-5 years to put out a truly new automobile! Of course, the fundamental level of complexity of the automobile was lower in 1935. However, Jenney and his coauthors give considerable credit to the small design team and the even tinier office they occupied. They were practically sitting in each other's laps – team communication was not an issue.

Small software design teams are similar. A team of fewer than ten software engineers will usually be very quick and effective at putting together good products. If they sit together while they are designing, communication will be lightning fast. Very little formal process will be needed. As long as it is clear what the team needs to implement, a small team will be the most efficient way to get it done.

C. Sometimes Starting Small is Not an Option

The most popular response to this problem has simply to advocate the avoidance of building anything large. The Agile movement [5] advocates informal, verbal approaches and rapid iterations of small increments of working software. The "low ceremony" approach advocated by the Agile community is a perfect approach when you are designing something like a social media application. No one really knows what the users want. Trying to write a detailed design specification for a new social media application would be a waste of time. The best approach is simply to code the smallest possible thing up, put it out there, and see how people respond.

Unfortunately, not all problems fit this sort of unstructured, small increment, approach. It is pretty difficult to deliver "part of" an aircraft carrier. Even such benign products as a car are difficult. Davey [6] explains that a typical vehicle now has 50,000 baseline mechanical requirements and when these articulated down to the 50-70 onboard computers, they become 450,000 extended requirements. Even worse, the market is very sensitive. Even consumers who are not engineers are likely to notice if the manufacturer skips any significant set of the 50,000 baseline mechanical requirements. Again, it is basically impossible to "iteratively develop" something of this complexity.

D. Partitioning as a Complexity Management Strategy

I propose an alternate strategy for situations in which the software application to be developed is large: just chop the application up into a series of smaller, autonomous software subsystems that can be developed by small, independent teams. This strategy sounds simple, even facile. However, there is some subtlety to the approach.

Most important is that the separation between the pieces has to be strong and deep. The interfaces between the pieces have to be clear and unambiguous. They also have to be rigid. In many cases, it may actually be better to have separate companies develop the different pieces. Simply dividing a large team into a lot of departments and giving each department a name for part of the system is rarely sufficient. Why don't department naming strategies work? The reason is easy: software engineers are lazy. If all 100 software engineers have access to the same source code, the fact that they are divided into 10 different notional teams will have no effect on behavior. All 100 software engineers can and will find ways to save effort by putting hooks and links back and forth all across the system.

Doesn't object orientation solve this problem? Unfortunately, it doesn't. If all 100 software engineers have access to the entire source code base, you will simply end up with an object-oriented mess instead of a procedural mess.

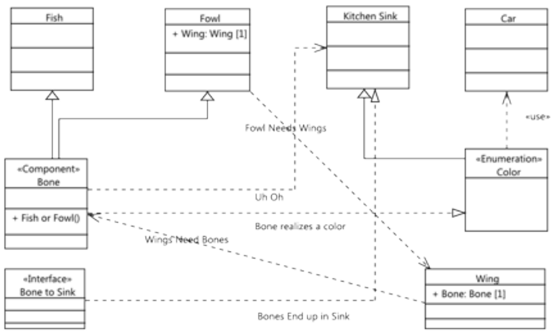

Figure 2 – Object-Oriented Mess

By partitioning the system rigidly and ruthlessly, you force each team to focus on the interfaces available and create something that works in a loosely coupled manner.

E. Partitioning as an Availability Strategy

As I mentioned above, for this strategy to work, the partitioning needs to be robust. Separate machines would be ideal. If possible, you want each development team to have no access to the rest of the system other than the defined interfaces. Separate machines (even better: different programming languages) are an ideal mechanism to enforce this disciplined separation.

Interestingly, the imperative to break the application up into pieces in order to reduce the software development complexity is nicely aligned with the IT operational consideration of breaking things up into autonomous pieces in order to improve operational availability.

In fact, the pieces of the application don't even really need to be on separate machines in order to feel separate. With technology such as Docker Linux containers [7] can run on the same machine with very low management overhead while retaining a high degree of isolation and protection from each other.

F. Cloud and Mobile Demand Partitioning

One of the most common current user application configurations is something that runs in a cloud, is accessible from a browser, and is also accessible from an iOS or Android device. This sort of application is inherently a partitioned/distributed application.

II. APPLICATION PARTITIONING

If we are going to partition an application to reduce complexity and project risk, we need to focus on interfaces and requirements traceability. We also have to select a modeling tool/paradigm suited to the task.

A. Interfaces are the Key Issue

If we want to enable teams to operate autonomously, interfaces are the key success factor. As an example, we can examine the success of Universal Serial Bus (USB) and the enormous number of successful products that use USB. While it is certainly possible to make a custom USB protocol, a large variety of products simply emulate mass storage devices. The interface is simple, relatively easy to implement, stable, and above all rigid. That is, the mass storage device interface changes rarely if ever at all. Since it is more or less impossible to get custom modifications to the interface, developers don't even try. They simply figure out how to use it as it exists in order to make their individual product work.

Another key success factor for USB is the availability of a compliance testing ecosystem. Informal testing is quite easy since almost every notebook computer comes with at least one USB port. More detailed testing can be done at the periodic compliance workshops run by the USB standards organization [8] or by independent testing laboratories.

Note that it is not necessarily easy in every case to define a nice, crisp, granular interface for every technology. If you are using a technology like Java Server Pages (JSP) [9] considerable skill and ingenuity is required to define, model, and produce a test suite for the interface.

B. Requirements Traceability

Another key aspect of breaking up the software application and assigning the pieces to multiple autonomous teams is that requirements traceability becomes critical. That is, a strong central project team is required, and that team needs to be able to track that all the original source requirements and their derivatives are actually allocated to pieces of the partitioned software application.

C. What about UML?

The standard tool for modeling software applications is the Unified Modeling Language (UML) [10]

The initial design of UML was dominated by thinking about class hierarchies. If we examine a classic UML text like Booch et al. [11] more than 100 pages are devoted to class hierarchies and their relationships while only 10 pages are devoted to interfaces. This emphasis is hardly surprising as the authors (and key originators of UML) were also the leading early evangelists for the transition from procedural programing to object-oriented programming.

Unfortunately, class hierarchies are not a very useful paradigm for modeling distributed software applications. Consider the case of an iOS application that has to work together with a cloud-based application.

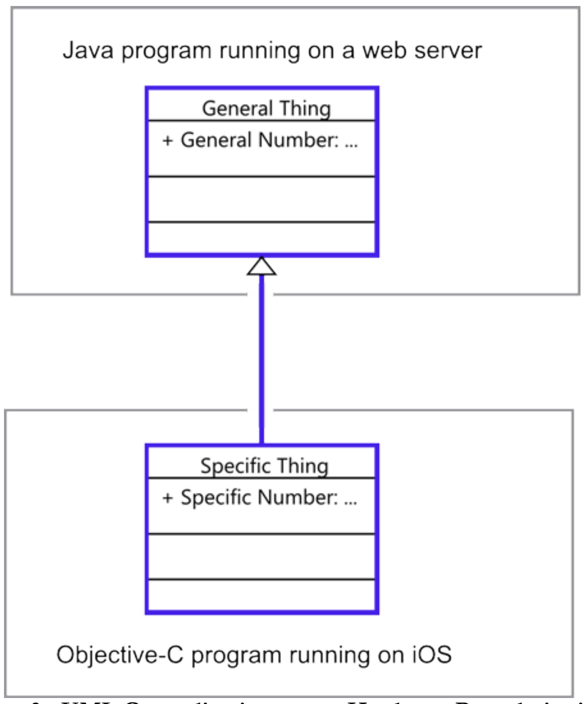
Java program running on a web server
General Thing
+ General Number: ...
Specific Thing
+ Specific Number: ...
Objective-C program running on iOS

Figure 3 - UML Generalization across Hardware Boundaries is Problematic

Modeling the system as a UML class hierarchy is going to be problematic. There really isn't a compiler technology that is going to allow an Objective-C class running on an iOS device to "inherit" its methods and properties from a Java class running on a cloud host. It is much more effective to start with a modeling approach that is focused on creating partitioned and distributed systems.

D. Advantages of SysML

Systems Modeling Language (SysML) [12] is an extension of UML developed jointly by the Object Management Group (OMG) [13] and the International Council on Systems Engineering (INCOSE) [14] to adapt UML to meet the needs of the systems engineering community.

The main benefits of using SysML rather than UML for modeling partitioned/distributed software applications come from three main improvements introduced in SysML:

1. Increased Focus on System Partitioning – The original definitions of UML did not provide strong support for system partitioning. Authors of key early textbooks included comments like: "All bets are off if distributed processing is involved"
2. Improved Interface Support – The SysML definition and community of practice has put a lot of effort into interfaces. While UML did have interfaces, the same early UML textbooks contained admonitions like: "Don't pay too much attention to interfaces because they are going to change anyway"
3. Requirements Traceability – This function was simply missing from UML.

III. Proposed Methodology

The proposed SysML-based software partitioning methodology is as follows:

A. Solid Use Case and Requirements Foundation

The foundation is a solid implementation of use case development as per Cockburn [15]. However, as Cockburn points out, UML (and by extension SysML) really only models the existence of use cases. The content of the use cases is a text artifact. The semantics of the use case content is not directly represented in a UML (SysML) use case diagram.

As such, the next step is to derive specific formal requirements from the use cases. For this we use a more formal requirements process such as the one described by Wiegers and Beatty [16] to develop atomic requirements. Note that in practice the use case activity [15] and the requirements definition activity [16] will overlap and some adjustments will be needed.

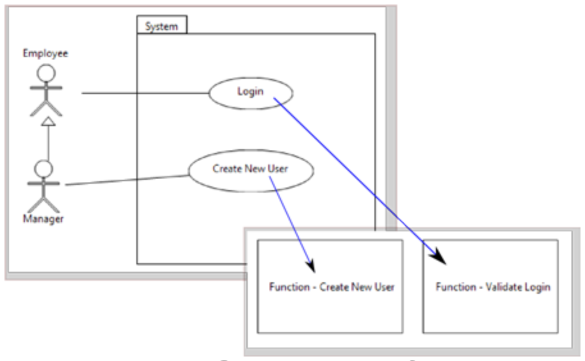

Figure 4 - Requirements Derived from Use Cases

In principle, the requirements can be managed directly with a SysML tool. However, for a large and complex software application, it will usually be easier to use a specialized requirements management tool that integrates with the SysML modeling tool.

B. Partition for Small Team Implementation

From the use cases and requirements, you can begin to make the next three pieces:

1. Logical Architecture – for example: "Model, View, Controller"
2. Physical Architecture – for example: "Cloud implementation hosted by XYZ.com, iOS user application, Android user application, Browser user application"
3. Sequence Flows – For each use case, develop the flows and sequences between the physical architecture pieces as required to support the use case and the requirements derived from the use case.

One would wish to be able to describe the steps above as a straightforward, linear, sequential process. Unfortunately, considerable iteration may be required to reach a combination of the three above that is coherent and meets the objectives.

Many different factors will need to be considered including:

1. Reuse of existing software services or components. We don't want to waste effort reinventing something when an adequate component or service already exists. However, it will be important to establish whether the existing software has a well-defined or at least easy-to-characterize interface.
2. Size of each piece. Our goal is to keep each new piece to be developed to something that a 5-10 engineer team can handle with reasonable risk. However, figuring out what is "small enough" is will require considerable skill and experience with similar projects. It may be possible to gain intuition from the requirements.
3. Non-functional requirements. Use cases are extremely important. However, some portion of the requirements in a realistic system will come from government regulations or other sources that may not directly be related to an actual use case.
4. Understandable architecture. We want both the logical and physical architecture to be as easy to understand as possible. Simplicity is important because we are going to have to derive interface behaviors that reflect the architecture.
5. System context. It will be important to determine the system boundaries because any flow across a boundary will need to be supported by a characterized interface. Determining the boundaries is not as easy as it sounds. For example, in an iOS application, the iOS operating system itself is definitely not within the scope of your application. As such, you need to identify the interfaces needed to the iOS operating system such as local storage, display, user input, and so on.

C. *SysML Requirements Trace and Allocation*

At least by the end of the previous step, we will want to use the SysML requirements trace and allocation capabilities to make sure that all of the original requirements are addressed by the partitioned system.

D. *Interface Definition*

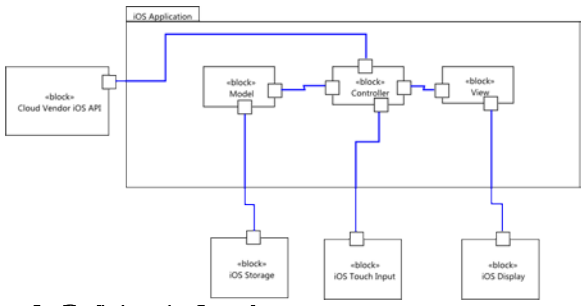

Figure 5 - Defining the Interfaces

We next need to identify all the interfaces. For each interface, we should identify the sequence flows that apply above and make submodels of the exact sequence flow across the specific interface. For example, a given sequence flow from a use case might travel through 6 subsystems and across 8 interfaces including interfaces to things outside the system. In this case, 8 subsequences should be modeled, one for each interface.

E. *Model the Integration Test Framework*

The key step that will make the partitioning successful is to be able to model and hopefully partially generate an integration test framework. In a general sense, we need to provide the autonomous development teams two emulators for each interface.

Consider a portion of the system in which two parts of the software are communicating via one of our interfaces:

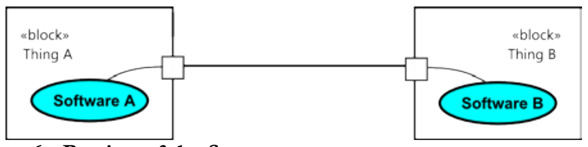

Figure 6 - Portion of the System

The first emulator we need to make will allow the team working on "Software A" to work in the absence of "Thing B"

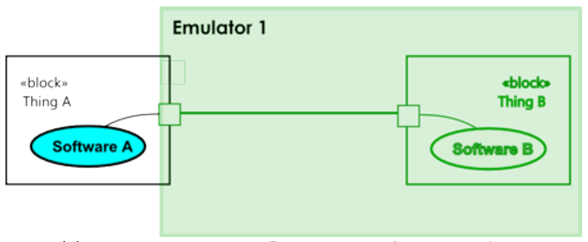

Figure 7 - Test Emulator 1 for Developing Software A

The second emulator will work the other way around:

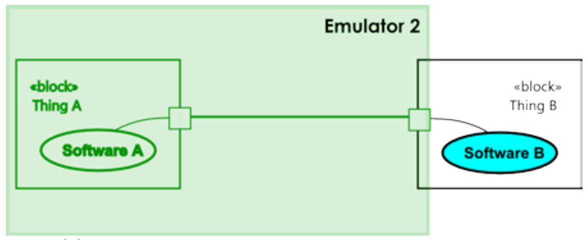

Figure 8 - Test Emulator 2 for Developing Software B

Where one side of the interface is out of the system scope, we only need to provide the emulator for the side of the interface that is within the system scope.

IV. CONCLUSIONS AND FURTHER RESEARCH

During the Fall of 2013, Asatte Press will be testing the proposed methodology by implementing a simple customer relationship management (CRM) application using the Google App Engine as the cloud host and providing user interfaces for a web browser, iOS, and Android. The key focus of the continuing Asatte Press effort is to determine the extent to which working test simulations can be generated from the SysML models and whether each piece can in fact be developed in isolation using those test simulations. While final results will not be available until early 2014, initial study of the work of the Test Driven Development (TDD) [17] community indicates that use of off the shelf test emulation technology such as JMock [18] should allow at least a useful partial generation of the simulators needed.

REFERENCES

[1] Frederick P. Brooks Jr., The Mythical Man Month: Essays on Software Engineering, Addison Wesley, USA, 1975.

[2] Kaled El Emam, and A. Günes¸ Koru, "A Replicated Survey of IT Software Project Failures," IEEE Software Sep/Oct 2008 pp 84-90.

[3] Joe Jenney with Mike Gangl, Rick Kwolek, David Melton and Nancy Ridenour, Modern Methods of Systems Engineering: With an Introduction to Pattern and Model Based Methods, 2010.

[4] Image from Wikimedia Commons by Cliff in Arlington, VA : http://commons.wikimedia.org/wiki/File:1936_Cord_810_Phaeton_%282835393420%29.jpg - Checked 30 September 2013

[5] http://agilemanifesto.org/ - Checked 29 September 2013

[6] Christopher Davey, "Automotive Software Systems Complexity: Challenges and Opportunities" 2013 INCOSE International Workshop, 26 Jan 2013

[7] https://www.docker.io/ - checked 29 September 2013

[8] http://www.usb.org/developers/compliance/ - checked 30 September 2013

[9] http://www.oracle.com/technetwork/java/javaee/jsp/index.html - check 30 September 2013

[10] http://www.uml.org/ - Checked 30 September 2013

[11] Grady Booch, James Rumbaugh and Ivar Jacobson, The Unified Modeling Language User Guide, Second Edition, Addison Wesley, USA, 2005

[12] http://www.omgsysml.org/#What-Is_SysML – checked 30 September 2013

[13] http://www.omg.org/ - checked 30 September 2013

[14] http://www.incose.org/ - checked 30 September 2013

[15] Alistair Cockburn, Writing Effective Use Cases, Addison Wesley, USA, 2001

[16] Karl E Wiegers and Joy Beatty , Software Requirements, Third Edition, Microsoft Press, 2013

[17] http://www.agiledata.org/essays/tdd.html - checked 30 September 2013

[18] http://jmock.org/ - checked 30 September 2013