Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 28 (2014) 422 – 429
Procedia Computer Science

Conference on Systems Engineering Research (CSER 2014)

Eds.: Azad M. Madni, University of Southern California; Barry Boehm, University of Southern California; Michael Sievers, Jet Propulsion Laboratory; Marilee Wheaton, The Aerospace Corporation
Redondo Beach, CA, March 21-22, 2014

High profile systems illustrating contradistinctive aspects of systems engineering

Adam Burbidge*[a], Larry Doyle[b], Michael Pennotti[a]

[a]*Stevens Institute of Technology, Castle Point on Hudson, Hoboken NJ, 07030-5991, USA*
[b]*ITT Exelis, 77 River Rd, Clifton NJ, 07014, USA*

Abstract

Many modern systems have a high degree of dependence on embedded software in order to perform their required functions. Some examples include transportation systems, hand-held devices, and medical equipment, among others. In designing their products, systems engineers typically take a top-down, process-oriented approach, decomposing a complex system into simpler, easier to manage, subsystems; the system requirements can then be allocated and flowed down as necessary to the appropriate subsystems. Software engineers take a more bottom-up, object-oriented approach, using simple building blocks to create a more complex system, and enhancing their existing blocks with new ones where necessary.

In many cases, both techniques must be employed together in order to design a successful system. Although it may have been acceptable in the past for simpler systems to view software as a separate subsystem with a fixed set of requirements, greater complexity of modern systems requires a corresponding improvement in working methodology. With the software playing an increasingly pivotal role, systems engineers must become much more familiar with the architecture of the software than previously; Likewise, software engineers need a systems-level view to understand which aspects of the design could be volatile due to new stakeholders (bringing with them new requirements), technology upgrades, and the changing world in general.

* Adam Burbidge. Tel.: +1-858-627-6860
E-mail address: aburbidg@stevens.edu

Selection and peer-review under responsibility of the University of Southern California.
doi:10.1016/j.procs.2014.03.052

Systems whose success or failure play out in the public arena provide a rare opportunity to study the factors that contribute to their outcome. Using two such systems, the Denver International Airport baggage handling system and the Apple iPad, this paper will study some best practices that can lead to project success or failure, and show the importance of a rigorous capture and flow down to both hardware and software of the requirements that must be correct from the start, as well as of designing an architecture that can accommodate the inevitable changes to a system.

Designing extensible systems with a tolerance for future changes is a key factor in modern complex systems. The baggage handling system failed in part because of a failure to appreciate the central role of software and an apparent lack of a suitable strategy for handling requirement changes. Methods for creating software which is resilient to change have been well studied; however what may be somewhat lacking even to the present day is a broader education of the existing body of knowledge, and how to integrate it with systems engineering methods.

The iPad succeeded where many of its predecessors had failed by a successful application of traditional systems engineering techniques and correctly implementing the hardware elements. Coming from companies with experience in software development, the system extensibility was not an issue in this case. However, the designers of the earlier systems seemingly failed to understand the actual market needs, failed to develop a corresponding set of requirements to meet those needs, and failed to translate those requirements into an integrated hardware/software solution.

Selection and peer-review under responsibility of the University of Southern California.

Keywords: embedded software; change tolerance; requirements management;

1. Introduction

As the products created by engineers become more complex, modern systems have an increasing dependence on embedded software in order to perform their required functions; examples include medical equipment, transportation systems, and hand-held devices. The engineering of these systems has two aspects distinguished by characteristics that appear to go in opposing directions (i.e. contradistinctive aspects). Traditional systems engineers typically take a top-down, process-oriented approach, decomposing a complex system into simpler, easier to manage, subsystems. The corresponding functional requirements can then be allocated and flowed down as necessary to the appropriate subsystems, with a growing number being allocated to software. In contrast, designers of software systems concentrate on how the system architecture can be constructed to accommodate inevitable change due to new stakeholder requirements, technology upgrades, and the changing world in general. This approach has evolved from software engineering, where software engineers take a more bottom-up, object-oriented approach, using simple building blocks to create a more complex system, enhancing their existing blocks with new ones where necessary. In many cases, both approaches must be employed together in order to design a successful system. Reconciling the differing approaches of systems and software engineers has been a subject of discussion for some time. This paper will consider two examples that illustrate how the systems and software engineering disciplines have evolved and matured over the past decades, while highlighting some of the issues that still remain.

(Maier 2006) states that "the traditional state of the practice in systems engineering and system architecture descriptions are often not well suited to support complex software developments. For example, traditional systems engineering practice is to take a function-first approach to definition,"[8] whereas "the most successful very large system architectures typically take a strongly layered approach, with deliberate selection of convergence layers, a concept not typically found in traditional systems engineering practice."[8] The paper suggests that, in an ideal case, the waterfall method of development might work, if each step could be completed before starting the next, but recognizes that in practice the design process is more iterative, because systems and software engineers must build of each other's work.

As a result of a growing number of functional requirements being allocated to software, there is an increasingly intimate connection between the embedded software and the overall system. (Doyle 2006) mentions that for earlier

(that is, simpler) systems, it did not matter as much what approach (i.e. function-first or object-oriented) was taken with the software, because "the internal design of the software was not visible at the system design level".[15] The conclusion is drawn, however, that in more recent systems it is no longer adequate to view embedded software as simply a "black-box" sub-assembly, given that the level of interaction with the software has increased.

(Doyle 2004) further highlights that, since many of the requirements will be met by software, an understanding of the methodology is needed.[17] Indeed, (Turner 2010) indicates that not only a major portion of the capabilities, but also many of the more qualitative attributes, such as safety, security, and reliability, are largely dependent on software[10]; these attributes are discussed further in (O'Brien 2005)[12]. According to the hypothesis presented, the backgrounds of systems and software engineers may (at least partly) explain their preferences for certain tools. From (Dewar 2012), it would seem that there is also an unfortunate reality that many software engineering students are simply not being taught the tools they need to succeed.[23]

(Shaw 2010) suggests that "software engineering" as it currently exists, "lack[s] a firm engineering basis" and points out that, in fact, "software is often developed, modified, or tailored by people whose principal responsibility is not software development".[11] However, she states that software engineering *should* be an engineering discipline because, as defined at Carnegie Mellon, it is "the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind",[11] and the paper offers suggestions on how to increase the "engineering" aspect of "software engineering".

(Katic 1996) mentions that, "if the requirements change, a system based on decomposed functionality may require massive restructuring"[19], whereas the object-oriented method "focuses first on identifying objects from the application domain, then fitting procedures around them"[19], which actually tends to hold up better as the requirements evolve over time. From this, we can note that systems should be designed with a tolerance for change. That is, the requirements cannot all be known in advance, so the system needs to take into account some potential volatility as new stakeholders arrive or new information becomes available. Recognizing the need to reconcile the two approaches, (Hoffmann 2000) and others have described a model-based system and software development processes that combines both function-driven systems engineering and object oriented software engineering.[20] (Daniels 2004) describes a hybrid process to combine traditional requirements and use cases.[18]

2. Case studies

The developments of the baggage handling system at the Denver International Airport and the Apple iPad demonstrate how each approach can be critical to success or failure. These approaches represent two contradistinctive aspects of systems engineering: the capture and flow down of requirements to drive features that cannot be changed, and designing system architectures that are resilient to some inevitable change. In the one case, the project succeeded by focusing on the former, whereas the other failed due to lack attention to the latter.

2.1. The baggage handling system at DIA

In the 1990s, the new airport at Denver became synonymous with failure, particularly when an impromptu media event (made "without first informing [the contractor]"[2]) publicly demonstrated the flaws in the software controlling the still-incomplete baggage handling system. Much has been written about this airport, and it is worth noting that software was not the only problem; there were also many other issues ranging from the selection of the site to the design of the concourse buildings. However, the problems with the baggage handling system and its embedded software unintentionally placed this element on the "critical path" of the schedule, and one of the key reasons for the delay in the ultimate opening date.

When first proposed, the automated baggage handling system may have seemed like a relatively straightforward upgrade from other similar systems. Systems engineers could simply modify requirements from existing systems.

Likewise, software engineers could build on existing software, and scale it up for the new system. In practice, however, it is rarely, if ever, so "easy" to reuse an existing system; different circumstances, with different stakeholders operating under different conditions, almost inevitably lead to at least some changes in the original design. It is, therefore, critical to employ a design methodology that is tolerant of some degree of scope creep.

One of the problems with the baggage handling system was a failure to appreciate the central role of software. To those not working in the field, it may have seemed incredible that a "simple" software problem could postpone the opening date of an entire airport. One commentator referred to the software engineers as "Lilliputians", implying that software is such a small and insignificant thing that it shouldn't possibly require as much time and money as it does to complete a project.[7] This view is not unique: (Kerzner 2009) commented on the frustration that "a mere software failure" could delay the entire airport's schedule, again implying that software is simple and the problem should have been easy to fix.[25] Yet another contemporary article explicitly referred to the DIA system as "a laughing matter", boldly stating further that "everyone, even the most cynical technophobe, feels that this kind of task shouldn't be all that difficult [for computer software to accomplish]".[24] The reality is, however, that software projects in general are neither "small" nor "simple": the baggage handling system in particular was massively complex, much more so than any other similar system that had been built to date.

At the time, the engineers involved did not fully understand software architecture or the importance of properly managing changes. The initial strategy attempted on the DIA system to limit scope creep was to "freeze" the requirements, and not allow any changes. Despite being one of the primary active stakeholders, airlines were initially excluded from discussions about the baggage system's design[1]. When they were finally brought on board, they unsurprisingly requested many changes, which the development team was ill-equipped to handle. In the years since, we have learned that instead of freezing the requirements entirely, a better strategy would be to identify key requirements that must be available at launch, and which ones can be postponed until a later update.

Likewise, software engineers did not yet understand how to create software that is resilient to changes, and scaling the project up from existing systems proved to be more difficult than anticipated; the complexity has been compared to "4000 taxicabs in a major city, all without drivers, being controlled [remotely] by a computer through the streets of a city".[25] (de Neufville 1994) gives a rule of thumb that "if the system is 10 times as complex, the difficulties could be 100 times as great"[2], and further estimates that the automated baggage system planned for Denver would be about "100 times as complex as comparable systems elsewhere".[2] This underestimation of the complexity led to the baggage handling project being started later than it should have, meaning that there was not enough time to complete the work.

While many things have changed in the intervening years, one take away from this example is that the general public (and the media in particular), and even some of those working in the industry, tend to grossly underestimate the difficulty involved in creating embedded software in a complex system. Changes to the initial design are all but inevitable, so an architecture that can tolerate changes and a suitable strategy to handle them are critical.

2.2. The Apple iPad

From the timeline presented in (Shaw 2006), the state of software engineering matured in the years between the DIA and iPad projects.[13] As technology progressed, several companies, Microsoft notable among them, attempted to develop tablet-style computers. However, despite pioneering the field, Microsoft struggled to create a lasting impact on the market. An early attempt at creating a tablet computer was the PaceBook, released in 2002 from manufacturer PaceBlade, and running an edition of Microsoft's operating system.

By this time, Microsoft already had several years of experience with software development to become familiar with the need for system extensibility. However, in conjunction with its hardware partners, it seemingly failed to understand the actual market needs when designing the PaceBook. The device was reviewed in depth by (Witheiler 2002), and the reviewer notes that while there were many innovative design features to show that the designers were

thinking about real-world uses, there were also some limitations. For example, the reviewer notes that the size and weight of the device were not conducive to its intended use as a tablet. The engineers may have recognized the need for the device to be small and lightweight, but seemingly failed to translate this into a corresponding requirement; doing so may have led them to make different design choices.

There were also some aspects of the hardware/software interface that made the PaceBook less appealing than it might otherwise have been. For example, users of Microsoft's operating system were familiar with right-clicking the mouse to perform certain tasks. On the PaceBook, the item first had to be selected with the stylus, and then a separate button on the device pressed. The reviewer suggested it would have been more intuitive to include a button on the stylus itself; there were also some other examples that suggested the designers were limited by the technology available at the time.[3] The hardware and software elements were at different levels of maturity, and product analyst Paul Jackson commented "Software engineers got ahead of the hardware capabilities".[5] In short, ultimately the systems engineering failed, being unable to produce an integrated hardware/software solution that would actually appeal to consumers; many of the deficiencies were hardware issues that could not be addressed with a simple software update. The PaceBook was just one example, but other products from the same era faced similar challenges, and the tablet market remained lukewarm for several more years until the introduction of the iPad.

In contrast to its predecessors, the success of the iPad seems to be a triumph of traditional systems engineering techniques. Recognizing a need in the market, the designers were able to both build on existing technology and take advantage of new developments. With a better assessment of the requirements for the user interface, the iPad development team were able to make some trade-offs that might not have been obvious otherwise. For example, while many of the earlier tablets attempted to include features typically found on a desktop or laptop computer, Apple deliberately omitted several features. Apple already had an operating system that could be adapted for the new product, and technological advances in the hardware also allowed the iPad to be smaller and lighter than its earlier counterparts. Although previous tablet-style products failed to impress consumers, better technology and improved systems engineering techniques helped lead the iPad to success where its predecessors had failed. In fact, not only did customers buy (and continue to buy) the iPad in ever-increasing numbers (with about 3.3 million sold in its first quarter, and over 100 million after 2 1/2 years, according to numbers released by Apple[16]), but they actually bought them in addition to the music players, smart phones, and computers that they already had.

Reuse of a familiar interface was one factor in the iPad's success; with an operating environment nearly identical to the earlier iPhone, one of the marketing ploys was to say "you already know how to use it".[22] While the PaceBook also used a familiar operating system, some non-standard interactions meant that users had to learn new methods to perform certain tasks. By avoiding some of these non-intuitive design decisions, the iPad provided a better cognitive fit because its interface matched the user's existing mental models.

It should be noted that the complete iPad "system" covers more than just the device itself: there are also the content providers, the developer kit for third-party applications, the system for delivering software updates, and Apple's customer support system, as well as several peripheral devices produced by Apple and other vendors. Functions and features not available "out of the box" can often be added in later with external devices or software upgrades. These aspects are all important for updating the iPad, extending its capabilities, and keeping it relevant in a changing environment.

Although much of Apple's development process is secret, (Panzarino 2012) notes that there are weekly design meetings to discuss all products in development.[4] Whatever the details of their process might be, they were successful in identifying features that could not be fixed with a simple software upgrade, and implementing them correctly from the beginning.

3. Implications for best practices in systems engineering

These case studies call particular attention to some best practices the engineering of complex systems. While obviously not a comprehensive list, each represents an important consideration in the design of a complex system.

Embedded software in modern systems cannot be treated as just another subsystem. In the DIA case, the project was attempted in an era when the importance and complexity of embedded software was not yet fully understood, and it was this failure to appreciate the potential difficulties that led to some of the problems encountered.

The property of change tolerance "is of paramount importance in complex emergent systems"[9]. In the design of a complex system, it is important to consider all the stakeholders, especially during the requirement definition stage. The DIA project initially omitted a key set of stakeholders. But, there were circumstances that made it impossible to involve, or even identify, all the stake holders critical to the acceptance of the system at the outset. In addition, (Ravichandar 2007) identifies other sources of changes on a complex project: "varying expectation of the stakeholders, changing user needs, technology advancements, scheduling constraints and market demands"[9]. In the DIA baggage system, the designers attempted to handle the complexity by defining the requirements up front, and then setting a baseline and not allowing them to change. As described in (Ravichandar 2007): "Traditional Requirements Engineering (RE) strives to manage volatility by baselining requirements. However, the dynamics of user needs and technology advancements during the extended development periods of complex emergent systems discourage fixed requirements."[9] Tactics for designing systems that facilitating evolving designs have been studied by Bachman et al[21].

The development methodology must also accommodate change. In the years since the DIA debacle, system design has shifted to a more iterative process. (Boehm 2005) suggests that "the traditional sequential approach to software development [that is, the waterfall method, although the term is not explicitly used] [...] is increasingly risky to use"[6], and that it is preferable to have systems and software engineers working concurrently to design the final product. (Maier 2006) further states that "In a spiral development complete systems engineering cannot, even in principle, take place until long after the software architecture has been established. After all, if it were possible to complete all aspects of systems engineering early there would not be a need for the spiral."[8] Of the "ilities", modifiability is one of the key characteristics to allow a design to adapt to change requests and new requirements that may have been initially unknown; (Bachmann 2007) discusses tactics for handling modifiability in the architecture, and raises four major concerns when dealing with the modifiability of a software architecture: "1) Who makes the change? 2) When is the change made? 3) What can change? and 4) How is the cost of change measured?"[21] The answers to these questions emphasize the iterative nature of the design process, and help to understand and lessen the impact of the changes.

It is important to identify requirements that cannot be addressed with a software upgrade. The story of the tablet computer shows that some requirements drive design decisions that will be very hard to change in later stages of development, and these requirements must be captured correctly from the outset. This is frequently the case with systems that must interact directly with the physical realities of size, power, weight and man/machine interface devices. Requirements such as these are a counterpoint to requirements that can be implemented with change tolerant software.

It is also important to recognize that the design of change resilient architectures and the rigorous flow down of critical requirements involve different cognitive styles[17]. According to (Ravichandar 2007), "our cognitive ability to examine a problem from both a top-down and a bottom-up perspective facilitates the application of widely diverse solution approaches."[9] Achieving both resiliency to change on one hand and rigorous capture of requirements for immutable system attributes on the other requires a synthesis of approaches. The two projects discussed here illustrate where this synthesis worked and where it didn't.

4. The education of systems and software engineers

The available literature reveals a broad range of discussion of systems and software engineering techniques, and methods to create modifiable and adaptable architectures. However, what may be lacking is a broader recognition of, and education about, the existing body of knowledge: how, for instance, a systems engineer develops systems thinking to become a systems engineer, and what skills a software engineer needs in order to be competent in the field. There may also be a perception that software engineering is "easy" (such that any engineer could do it, even without specific formal training), and a lack of recognition of what it is that a systems engineer actually does. This can lead to an erroneous conclusion that the necessary skills will be acquired "automatically" from prior work and on-the-job training, and perhaps also that certain individuals are more "suited" to work in a certain field than others. While it may be true that an experienced engineer will acquire new skills over time, a better understanding of the way engineers think and work should help to accelerate the process.

According to (Davidz 2005), the development of systems thinking does not occur "automatically", nor is it necessarily rapid: "some systems leaders believe it may take fifteen to thirty years to develop a senior systems engineer".[14] In order to reduce this time, the paper discusses "enablers" for the development of systems thinking. (Although not explicitly discussed, a software engineer's development would also be aided by similar enablers.) At first glance, it may seem that many of the enablers mentioned are innate characteristics, but upon closer inspection, it turns out that most of them could in fact be acquired with the proper training. The paper does warn, however, that the training techniques employed by most businesses were ranked poorly by the trainees in terms of their effectiveness, and tries to suggest ways in which the money and resources could better be spent.

Even before entering the workforce, however, there may also need to be changes and improvements at the school level when students are learning about systems and software methodology. According to (Dewar 2012), software education in the US, as it currently exists, by and large does not prepare students for the situations they may encounter in the "real world".[23] In part this is due to the nature of the classroom setting, which tends to discourage teamwork to facilitate tracking individual development, and often lacks sufficient time to pose large-scale, "interesting" problems. (Dewar 2012) concludes with some suggestions on how to improve, but the reality is that schools and businesses have only limited resources, and must make decisions based on what they believe is "best" to meet current needs. From reading these papers, however, the impression one gets is that there is a disconnect between what schools and businesses believe is needed, and what the individuals undergoing the training feel is best to maximize and accelerate their learning. As systems evolve and become more integrated and software takes on an increasingly important role, it is becoming apparent that systems and software engineers cannot work in isolation, but must find ways to improve their skills in various areas ensure that the right talent is available for the future.

5. Summary and conclusions

Although other factors were involved in the success of the iPad and the failure at DIA, the aspects considered here were significant. In the case of the DIA, lack of resilience in the design and of a suitable strategy for dealing with the inevitable changes led to a cascade of problems. In the case of the iPad, the product developers succeeded in determining a market need, identifying an apparent gap in the existing product lines, and successfully developing and implementing a corresponding set of requirements. Compared to its predecessors, the iPad better assessed the actual user requirements, making trade-offs and design choices that might not have been obvious without strong systems engineering techniques.

The surprising aspect of these case studies is that the success of traditional systems engineering methods occurs on a project that at first seems to be a software-intensive system, and a failure of software-oriented systems engineering occurs on a system that at first seems to be hardware-intensive. This reinforces the view that systems and software engineers will need to continue working together and enhancing their skill sets. In the end, engineering is not a static field, and as the world continues to evolve, systems and software engineering techniques will likewise have to adapt in order to design the systems of the future. It is not expected that the work discussed in this paper will

be a complete solution to the problem of integrating systems and software engineering, but rather a building block on which the foundation of further discussion can be built. Potential next steps from here to continue development would be to build on this foundation and develop a methodology to integrate bottom-up and down-top approaches in a systematic way. As a starting point for further research, the two examples presented in this paper can be used as a lead-in to discuss how to select and integrate the two approaches.

References

1. Calleam Consulting Ltd. "Denver Airport Baggage System Case Study." *Why Projects Fail*. 2008. <http://calleam.com/WTPF/?page_id=2086>
2. de Neufville, R. "The Baggage System at Denver: Prospects and Lessons." *Journal of Air Transport Management*. Vol 1 No. 4 (1994): 229-236. <http://ardent.mit.edu/airports/ASP_papers/Bag%20System%20at%20Denver.PDF>.
3. Witheiler, M. "PaceBook PaceBlade: The Tablet PC Arrives." *AnandTech*. 17 May 2002. <http://www.anandtech.com/show/912>
4. Panzarino, M. "This is how Apple's top secret product development process works." *Next Web*. 24 Jan 2012: <http://thenextweb.com/apple/2012/01/24/this-is-how-apples-top-secret-product-development-process-works/>
5. Stone, B., Vance, A. "Just a Touch Away, the Elusive Tablet PC ." *New York Times* 4 Oct 2009, <http://www.nytimes.com/2009/10/05/technology/05tablet.html?_r=3&>.
6. Boehm, B., "Value-Based Software Engineering: Seven Key Elements and Ethical Considerations", USC-CSE-2005-503, February 2005. <http://csse.usc.edu/csse/TECHRPTS/2006/usccsse2006-640/usccsse2006-640.pdf>
7. Sommerville, I. "Systems Engineering for Software Engineers." *Annals of Software Engineering*. 6.1-4 (1999): 111-129. <http://dl.acm.org/citation.cfm?id=314621>
8. Maier, M.W. "System and software architecture reconciliation." *Systems Engineering*. 9.2 (2006): 146-159. <http://onlinelibrary.wiley.com/doi/10.1002/sys.20050/abstract>
9. Ravichandar, R., Arthur, J.D., Broadwater, R.P. "Reconciling Synthesis and Decomposition: A Composite Approach to Capability Identification." *14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. (2007) <http://arxiv.org/abs/cs/0611072>
10. Turner, R. "Sibling Rivalries: Reconciling Systems and Software Engineering ." *Systems and Software Conference*. Salt Lake City, Utah. 8 Apr 2010. Lecture. <http://www.dtic.mil/dtic/tr/fulltext/u2/a557392.pdf>
11. Shaw, M. "Research toward an Engineering Discipline for Software." *FSE/SDP workshop on Future of software engineering research*. (2010): 337-342. <http://www.nitrd.gov/subcommittee/sdp/foser/Research%20toward%20an%20Engineering%20Discipline%20for%20Software%20-%20p337.pdf>
12. O'Brien, L., Bass, L., & Merson, P. *Quality Attributes and Service-Oriented Architectures* (Technical Report CMU/SEI-2005-TN-014). Pittsburgh: Software Engineering Institute, Carnegie Mellon University. 2005. <http://www.sei.cmu.edu/library/abstracts/reports/05tn014.cfm>
13. Shaw, M, Clements, P.. *The Golden Age of Software Architecture: A Comprehensive Study*. Carnegie Mellon University, Feb 2006. <http://reports-archive.adm.cs.cmu.edu/anon/isri2006/CMU-ISRI-06-101.pdf>
14. Davidz, H.L., Nightingale, D.J., Rhodes, D.H. "Enablers and Barriers to Systems Thinking Development: Results of a Qualitative and Quantitative Study." *PROCEEDINGS CSER*. (2005): <http://sse.stevens.edu/fileadmin/cser/2005/papers/35.pdf>
15. Doyle, L. J, Pennotti, M. C., Impact of Embedded Software Technology on Systems Engineering, INCOSE 2006, The 16th International Symposium, July 2006.
16. The Associated Press. "Number of iPads sold by Apple by quarter." *Yahoo Finance* 23 Oct 2012, <http://finance.yahoo.com/news/number-ipads-sold-apple-quarter-201153619.html>.
17. Doyle, L., J, Pennotti, M., C., Cognitive Fit Applied to Systems Engineering Models, Conference on Systems Engineering Research, April 2004. <http://www.stevens.edu/sse/sites/default/files/Cognitive_Fit.pdf>
18. Daniels, T. B., "The hybrid process that combines traditional requirements and use cases", Systems Engineering, 14 Sep 2004. <http://onlinelibrary.wiley.com/doi/10.1002/sys.20013/abstract>
19. Katic, N., Nevstrujev, B., Vogel D., Pendergast, M. O. "Bridging the gap between structured requirements and object-oriented analysis and design", Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, pp. 525-35 vol.3, 1996. <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=493248>
20. Hoffmann, H., "From function driven Systems Engineering to object-oriented Software Engineering", I-Logix whitepaper, 2000. <http://www.incoseonline.org.uk/Documents/Events/SC2003/SC03_day2_slot6_hoffmann.pdf>
21. Bachmann, Felix; Bass, Len; & Nord, Robert. *Modifiability Tactics* (CMU/SEI-2007-TR-002). Software Engineering Institute, Carnegie Mellon University, 2007. <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8299>
22. EveryAppleAds, *Apple iPad ad - What is iPad (2010)* .Apple Computer. Web. Retrieved 5 Nov 2013. <http://www.youtube.com/watch?v=QG-c75dOaB8>
23. Dewar, R., "The education of embedded systems software engineers: failures and fixes", *New York University and Adacore*, 18 March 2012. <http://www.embedded.com/print/4238223>
24. Cohen, D., ed. "Striving for the Impossible." *PC Pro*. 10 Aug 1995: 170-175. Print.
25. Kerzner, H. *Project Management Case Studies*. 3rd ed. New York: John Wiley & Sons, Inc., 2009. 539-582. Print.