NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

EDUCATIONAL GUIDANCE ON EXTENSIBLE SOFTWARE DEVELOPMENT

by

Damon R. Alcorn

September 2018

Thesis Advisor: Thomas W. Otani
Second Reader: Paul C. Clark

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE EDUCATIONAL GUIDANCE ON EXTENSIBLE SOFTWARE DEVELOPMENT	**5. FUNDING NUMBERS**
6. AUTHOR(S) Damon R. Alcorn	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Software extensibility is a software engineering principle that characterizes how easily new features can be added to the software system by requiring no or minimal rewrite of existing code base. Software that is extensible leads to reduced development time, increased stability and security, and better support of software assurance and maintenance. Although it is critically important, the Department of Defense (DoD) utilizes software development documents that provide only limited information and guidance on software extensibility. Moreover, the software development processes supported by the DoD do not fully address Model-View-Controller (MVC), a design pattern that industry experts recommend for a higher degree of software extensibility. This thesis studies the design patterns and software extensibility in the context of the DoD software development environment with the case study on mobile application design. The design guidance on software extensibility developed in this thesis will integrate well with currently utilized DoD software development documents and processes.

14. SUBJECT TERMS software engineering, software assurance, software maintenance, extensibility, software design patterns, model, view, controller, Model-View-Controller, MVC	**15. NUMBER OF PAGES** 75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

EDUCATIONAL GUIDANCE ON EXTENSIBLE SOFTWARE DEVELOPMENT

Damon R. Alcorn
Civilian
BA, California State University, East Bay, 2004
MA, California State University, Sacramento, 2008
AS, Las Positas College, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2018

Approved by: Thomas W. Otani
Advisor

Paul C. Clark
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software extensibility is a software engineering principle that characterizes how easily new features can be added to the software system by requiring no or minimal rewrite of existing code base. Software that is extensible leads to reduced development time, increased stability and security, and better support of software assurance and maintenance. Although it is critically important, the Department of Defense (DoD) utilizes software development documents that provide only limited information and guidance on software extensibility. Moreover, the software development processes supported by the DoD do not fully address Model-View-Controller (MVC), a design pattern that industry experts recommend for a higher degree of software extensibility. This thesis studies the design patterns and software extensibility in the context of the DoD software development environment with the case study on mobile application design. The design guidance on software extensibility developed in this thesis will integrate well with currently utilized DoD software development documents and processes.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

CC	Common Criteria
DoD	Department of Defense
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
KA	Knowledge Area
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
NIST	National Institute of Standards and Technology
UI	User Interface
VIPER	View, Interactor, Presenter, Entity, Router

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM OVERVIEW

When it comes to their software, companies such as Apple and Microsoft encourage the use of the Model-View-Controller (or MVC) software design pattern, or one of its very similar alternatives. They do this by hosting online guides that argue the benefits of adhering to these patterns, including easier application modification, greater code reuse, and extensibility.[1]

Extensibility, which is the ability for software to easily incorporate additional functionally, is a tremendously valuable and powerful feature to have, especially given the natural limitations of program testing and the fact that the needs of users are rarely static—often even evolving in completely unforeseen ways given enough time. For one, systems can have their operational life span extended—possibly even indefinitely—as they adapt to new user needs and roles as they materialize. This fact alone can result in considerable cost savings for an organization, whether it be in terms of money, time, workforce energy, computing resources, or whatever else might need to be committed to a new project in which an application is developed from scratch. Second, extensible applications have a higher degree of improvability. By allowing for not only slight modification but also the complete addition of functionality, these applications are more flexible and adaptable when it comes to actions such as overcoming newly discovered security and stability issues, reducing or eliminating the effect of non-beneficial or detrimental qualities, and refining or increasing the effect of advantageous qualities. As a result, extensible applications often have fewer bugs, are more secure, and operate with greater efficiency and effectiveness. Third, since their code is often both highly modifiable and reusable, extensible applications can often be developed much faster than

[1] "Model-View-Controller," Apple, accessed May 22, 2018, https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html; "The MVVM Pattern," Microsoft , last modified February 10, 2012, https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10).

non-extensible applications, which tend to have code that is either not easily changed or reused, or both.

Clearly, given the Department of Defense's (DoD) desire to quickly and easily field cost effective, stable, and secure applications that provide a maximum operational advantage, it would seem that the DoD has an interest in developing extensible applications. Despite this fact, the DoD goes about the software development process in a manner that does not really seem to stress the importance of extensibility or offer much guidance or support in terms of implementing it. For one, documents used by the DoD regarding software development mostly discuss the concept of application extension as it pertains to the documentation of software requirements, barely touching on how to actually create extensible software or doing so only indirectly.[2] Second, instead of advocating the use of extensible-minded design patterns such as MVC or one of its alternatives, the DoD favors a process in which software design is the result of something called a "Design Definition."[3] While the Design Definition process does not preclude the use of an extensible-minded design pattern, it does not really encourage it either. This is because the process allows for—maybe even actively encourages—ad hoc changes to an application's utilized design or design pattern in response to issues encountered during development.[4] This is argued to be an advantage, resulting in greater project adaptability, as well as a development environment that is highly compatible with the popular Agile development approach.[5] But this argument seems to ignore the fact that allowing for and even encouraging deviations from time-tested and vetted design patterns can quickly result in the loss of certain valued design characteristics—such as extensibility.

---

[2] Pierre Bourque and Richard E. Fairley (eds.), *Guide to the Software Engineering Body of Knowledge, Version 3.0* (Piscataway, NJ: IEEE Computer Society, 2014), 1-5, 3-3, 3-6.

[3] International Organization of Standardization, *ISO/IEC/IEEE 12207: Systems and Software Engineering - Software Life Cycle Processes, First Edition 2017-11* (Geneva: ISO/IEC, 2017; Piscataway, NJ: IEEE Computer Society, 2017), 71. This document replaced the previous relied upon MIL-STD-498.

[4] Ibid.

[5] Ibid., 127.

In short, the DoD utilizes software development documents that cover the concept of extensibility somewhat ineffectively, while simultaneously relying upon a process that undermines the use of extensible-minded design patterns to help ensure extensibility.

B. PROPOSED SOLUTION

Seeing the potential development of non-extensible applications by the DoD as a problem, this thesis attempts to investigate a manner in which the DoD might effectively go about ensuring application extensibility. Of course, one way to do this would be to always strictly enforce an unchanging extensible-minded design pattern. That said, that solution would probably threaten the argued advantages of adaptability and flexibility supported by the Design Definition process and its allowance for a potential evolving design. That is why this thesis takes a different approach.

Looking at the MVC and MVC-like design patterns used by industry as a starting point, this thesis attempts to distill the key features shared by these patterns that result in extensibility. In brief, these key features seem to be the following:

- The first is the decomposition of code into modules. This results in code that is easier to read and work with. Issues can be localized, and debugging is thus simplified. In this way, code is made more modifiable by being modular.

- The second is the decoupling of those modules. This results in code that is more reusable. By reducing the ways in which the internal design and implementation of one module may be reliant on the internal design and implementation of others, it becomes possible to modify that module and often simply reuse the rest.

- The third is the organization of modules into a clear-cut, hierarchical structure. After decomposition and decoupling are done, this action is what seems to finally result in extensibility. Modules operate so that dependencies are limited and one-way. This makes it easier to add new application functionality by means of refactoring or replacing branch and

leaf modules within the hierarchy, where the effects of those changes are less likely to require extensive alterations to the rest of the application's code base.

Using these key features, this thesis formulates an easy to use guideline that highlights the various requirements of producing extensible code. This guideline can be used as an educational tool for developing extensible software or checking for extensibility in an existing code base. It includes the following criteria:

- As a first criterion, to produce decomposed modules, code that is logically related must be grouped together and separated out for unrelated code, forming clearly distinct modules with clearly defined roles and concerns, as well as limited tasks.

- As a second criterion, to produce decoupled modules, the internal design of one module must not be reliant on the internal design of another. If a module "knows something" about the public interface of another module, it should be the caller of that module. The callee should not need to know anything about its caller, beyond how to return messages to it. Additionally, interactions between modules should be kept to a minimum and as general as possible, with no messages or data being passed that could leak any information regarding module implementation.

- Finally, as a third criterion, to produce hierarchically organized modules, confusing or circular interactions and possible dependencies must be avoided. There should not be instances of two-way coupling—that is to say, no module should be both a caller and callee of another module. And functionality that is more likely to be replaced or extended should be pushed out into branch or leaf modules, while core functionality (i.e., that which is not likely to be changed) should be located in modules that are closer to the hierarchy's root.

As one implements software following these criteria, the burden of an application's design process will likely increase. That said, achieving the first criterion should result in the benefit of modular and therefore easier to work with code. Implementing the second should result in the benefit of reusable code. And achieving the third should result in the benefit of extensible code—the end goal of this guide.

In showing the application and utility of this guide, this thesis includes a case study in which a simple extensible iOS application is reviewed. While this application conforms to the MVC design pattern used by Apple, more importantly, it clearly highlights the presence of the key features needed for application extensibility.

In this end, this guide could potentially serve as a tool for DoD personnel overseeing application development or acquisitions. Through its use, an individual could gain greater awareness of the basic features that generally result in extensible code, increasing the likelihood that that individual will be able to gauge if an application is extensible or make recommendations on what changes should be made in order to achieve extensibility. All of this serves the goal of hopefully increasing the likelihood that the DoD produces more cost effective, stable, and secure applications that provide a maximum operational advantage. Additionally, use of this guide should not work against the adaptability or flexibility of the Design Definition process in the way that strict adherence to an extensible-minded design pattern might.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. THE DOD SOFTWARE DEVELOPMENT PROCESS

The DoD uses various documents to give structure to its application development processes. The core documents used include the following:

- ***ISO/IEC/IEEE 12207: Systems and Software Engineering – Software Life Cycle Processes***: This document "establishes a common framework for the software life cycle processes," which at a high-level includes the "acquisition, supply, development, operation, maintenance, and disposal of software systems, products and systems, and software portions of any system."[6] The purpose of this document is for essentially "establishing business environments, e.g., methods, procedures, techniques, tools, and trained personnel" capable of successfully and systematically developing or acquiring software.[7] This document replaced MIL-STD-498 (*Software Development and Documentation*).

- ***ISO/IEC/IEEE 14767: Systems and Software Engineering – Software Life Cycle Processes – Maintenance***: This document "describes in greater detail management of the Maintenance Process described in ISO/IEC 12207 ... to provide guidance for the planning for and maintenance of software products or services."[8] The reason for this separate document that expands on the content covered in the Maintenance Process section of ISO/*IEC/IEEE 12207* is due to the fact that "maintenance consumes a major share of a software cycle financial resources," making it "an important project consideration" that could have a proportionally high and

---

[6] Ibid, 1.

[7] Ibid.

[8] International Organization of Standardization, I*SO/IEC 14767, IEEE Std 14767-2006: Systems and Software Engineering - Software Life Cycle Processes - Maintenance,* Second Edition 2006-09-01 (Geneva: ISO, 2006; Piscataway, NJ: IEEE Computer Society, 2006), 1.

far-reaching impact on the development of an application and its success in the field.[9]

- ***IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK 3.0):*** This document "constitutes a valuable characterization of the software engineering profession.... as a mechanism for acquiring the knowledge you need in your lifelong career development as a software engineering professional."[10] This characterization includes Knowledge Areas (or KAs) that contain "generally accepted" and "generally recognized" knowledge and practices, which are "distinguished from advanced and research knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application)."[11] Additionally, they "are applicable to most projects most of the time, and there is consensus about their value and usefulness."[12] These KAs cover numerous aspects of software development, including software requirements, design, construction, testing, and maintenance to name a few.[13]

- ***NIST SP-800-53 (Rev. 4): Security and Privacy Controls for Federal Information Systems and Organizations***: This document "provides a holistic approach to information security and risk management," providing guidelines for selecting and specifying security controls for organizations and information systems supporting the executive agencies of the federal government."[14] It is used by concerned parties to help clarify the burden

---

[9] Ibid., vii.

[10] Pierre Bourque and Richard E. Fairley (eds.). *Guide to the Software Engineering Body of Knowledge. Version 3.0*, (Piscataway, NJ: IEEE Computer Society, 2014), xvii.

[11] Ibid., xxxi, xxxiii.

[12] Ibid.

[13] Ibid., xxxii.

[14] National Institute of Standards and Technology, *NIST SP-800-53: Security and Privacy Controls for Federal Information Systems and Organizations, Rev. 4, (*Gaithersburg, MD: National Institute of Standards and Technology, 2013), xv, 2.

of meeting certain requirements and recommendations specified in other security-minded standards documents, including *FIPS Publication 200* (*Minimum Security Requirements for Federal Information and Information Systems*) and *FIPS Publication 199* (*Standards for Security Categorization for Federal and Information Systems*), as well as map specified controls to international security standards such as *ISO/IEC 15408* (*Common Criteria*).[15]

- ***NIST SP-800-600 (Vol. 1): Systems Security Engineering Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems***: This document "addresses the engineering-driven actions necessary to develop more defensible and survivable systems—including the components that compose and the services that depend on those systems."[16] To achieve this end, it relies upon "well-established International Standards for systems and software engineering … and infuses systems security engineering techniques, methods, and practices into those systems and software engineering activities."[17] In the end, the overall goal of the document is the realization of "trustworthy secure systems that are fully capable of supporting critical missions and business operations while protecting stakeholder assets" and, as a result, it approaches "security issues from a stakeholder requirements and protection needs perspective."[18]

Used together, these documents can be thought of as giving the DoD software development process its basic structure. *ISO/IEC/IEEE 12207* and *ISO/IEC/IEEE 14767* can be used to outline the work environment, processes, and operational considerations

---

[15] Ibid., 2, xv.

[16] National Institute of Standards and Technology, *NIST SP-800-600: Systems Security Engineering Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems,* Volume 1, (Gaithersburg, MD: National Institute of Standards and Technology, 2018), viii.

[17] Ibid.

[18] Ibid.

that an application's development will entail. IEEE's *SWEBOK 3.0* can be used to give guidance on the general knowledge and recognized practices that may be relied upon in each of the outlined processes. *NIST SP-800-53* provides specific guidance on issues of risk and responsibility that translate into security-minded requirements for software, software development processes, and certain parties developing software for organizations such as the DoD. And *NIST SP-800-600* provides guidance on the standards, methods, principles, and techniques that can be leveraged in order to enhance the security and trustworthiness the software being developed.

B. NOTIONS OF EXTENSIBILITY IN DOD SOFTWARE DEVELOPMENT DOCUMENTS

As stated before, extensibility (which, again, is the ability for software to easily incorporate additional functionally) is a tremendously valuable and powerful feature to have. It can translate into applications that can often cost less over their life cycle, can be more quickly developed and maintained, and exhibit greater stability, security, and functionality as a result of their higher degree of improvability. One would therefore think that the DoD would be highly interested in developing and fielding extensible applications; however, the documents described earlier strangely do not do a great job of addressing the concept at all.

For one, much of the discussion of extensibility is limited to the notion of documenting software requirements. In *SWEBOK 3.0*, it is stressed that the process of refining software requirements should include "a description of the software being specified and its purpose," but not one that is overly exclusive.[19] Instead it should be "scalable and extensible to accept further requirements" as they become known to stakeholders.[20] Alternatively, in *ISO/IEC 15408* (*Common Criteria*, which is referenced in *NIST SP-800-53*), it is explained that because "the understanding and needs of consumers may change," the functional requirements specified in the Common Criteria (or CC) framework "will need to be maintained," alluding to the possibility of updates, as well as the

[19] Bourque, *Guide to the Software Engineering Body of Knowledge*, 1-5.

[20] Ibid.

limitations of the current framework and the software security requirements, properties, and controls the framework may be used to specify.[21] The document then goes on to suggest that authors of Protection Profile and Security Target documents (i.e. CC framework documents outlining security requirements and properties) who "may have security needs not (yet) covered by the functional requirement components in CC…. may choose to consider using functional requirements not yet taken from the CC."[22] Within the context of the Common Criteria, this inclusion of additional requirements that are not yet covered by the framework is "referred to as extensibility."[23] In many ways, it is true that the addition of functionality to software can open the door to new threats, creating the need for new security controls and functional requirements. Therefore, it is understandable that the extension of an application could have a significant impact on the documented software requirements of that application. But that being said, extending software requirements is probably the easiest of the actions associated with software extension, especially when compared to the actual work of designing, constructing, integrating, and testing the extensions. As a result, it seems somewhat myopic that this is a significant portion of these documents' discussion of the concept of extensibility.

Second, any discussion of actual software extension by the documents—that is to say, how to go about software extension, or somehow recognize or ensure that it is possible with regard to an application's code base—seems superficial and not particularly useful. In *ISO/IEC/IEEE 12207*, the "extension of capability, mid-life upgrade, or evolution of legacy systems" are mentioned as common realities of the software maintenance process, often resulting from things like "changes to interfaced systems or infrastructure, evolving security threats, and technical obsolescence of system elements and enabling systems over the system life cycle."[24] But even with that said, the document gives no real guidance on how to actually go about anticipating or planning for software extension, or

---

[21] International Organization for Standardization, *ISO/IEC 15408: Common Criteria for Information Technology Security Evaluation - Part 2: Security Functional Components*, Ver. 3.1, Rev. 5 (Geneva: ISO/IEC, 2017), 17.

[22] Ibid.

[23] Ibid.

[24] *ISO/IEC/IEEE 12207,* 95.

how to make certain that extensions to an application's code base will be possible when the need arises. One might assume that such information is covered by one of the KAs in *SWEBOK 3.0*; however, its guidance seems fairly limited and not very useful as well. In it, it is stated, "Anticipating change helps software engineers build extensible code, which means they can enhance a software product without disrupting the underlying structure."[25] The book then promises that the action of anticipating change "is supported by specific techniques" listed in a later section.[26] Unfortunately, the actual nine techniques that *SWEBOK 3.0* lists are fairly high-level and not at all exclusive to the creation of extensible code, including rather basic and somewhat vague recommendations like "creating understandable source code, including naming conventions and source code layout"; the use of "classes, enumerated types, variables, named constants, and other similar entities"; the inclusion of documentation; the use of "control structures"; and "code organization (into statements, routines, classes, packages, or other structures)."[27] While it is good to see that these documents at least at some level inform readers about when software extension efforts might take place and that there are techniques for making such work easier when they occur, the provided information is fairly general and not particularly useful when it comes to practical application.

In truth, when it comes to these documents, perhaps the best source of practical information for use in the development of extensible applications is *NIST SP-800-600*, albeit somewhat indirectly. Given the fact that extensible applications are generally more secure than non-extensible applications, it makes sense that some of the concepts and practices used to engineer secure and trustworthy systems are also those used to develop extensible software. That is to say, in attempting to build a secure system, one may very well build a fairly extensible one as well, and vice versa. This is especially true when considering the "*structural design principles*" overviewed in Appendix F of the document, which cover "how the system is decomposed into its constituent *system elements*," as well as "how those elements relate to each other and the nature of the

---

[25] Bourque, *Guide to the Software Engineering Body of Knowledge*, 3-3.

[26] Ibid.

[27] Ibid., 3-6.

interfaces between elements."[28] Many of these principles—including the use of "*modularity* and *layering*... derived from functional decomposition," "*clear abstractions*" (i.e. "well-defined interfaces and functions"), "*partial ordered dependencies*" that result in linear inter-layer interactions as opposed to possibly circular ones, and "*reduced complexity*" by keeping systems "as simple and small as possible"—are leveraged to increase the "simplicity and coherence of the system design" with the aim of making the work of secure system analysis, development, debugging, and validation faster, easier, and less error-prone.[29] As discussed in later chapters, very similar principles are leveraged to also make simpler and more coherent systems that are faster and easier to modify, reuse, and extend. However, this fact is not expressly stated in *NIST SP-800-600* and the overall focus and drive of the document is the development of secure and trustworthy systems, not necessarily extensible ones.

In the end, the concept of extensibility is not completely overlooked by the documents used by the DoD for software development. That said, the discussion does seem fairly limited in scope, superficial, or somewhat indirect in its approach to the topic, pointing to the fact that the documents are probably not a sufficient source of education or guidance—especially for non-technical personnel, who within the DoD environment may very well be involved in a project's decision making process—on how to possibly implement or check for extensibility in software that is being developed for or by the DoD.

C. FURTHER NEED FOR GUIDANCE ON EXTENSIBILITY

The fact that documents relied upon by the DoD for the purpose of software development seem to insufficiently address the concept of extensibility is strong motivation for this thesis. Further motivation comes from the DoD's use of a design process that makes simple adherence to extensible-minded design patterns such as MVC potentially difficult.

---

[28] *NIST SP-800-600* (Vol. 1), 205.

[29] Ibid, 205-207.

As mentioned in the introduction, in order to give instruction and guidance to developers on how to create extensible code, companies such as Apple and Microsoft host online guides on the application of the MVC or MCV-like software design patterns. The same is not done by the DoD, which does not encourage the use of any particular design pattern. Instead, it favors the use of something called a "Design Definition" process.[30] In its most basic form, this process is intended to maximize project adaptability and responsiveness, allowing for changes to and refinements of a software design in response to decisions regarding desired and feasible design principles ("including controlling ideas such as abstraction, modularization and encapsulation, separation of interface and implementation, concurrency, and persistence of data"), security considerations (e.g. "the principle of least privilege, layered defenses, restricted access to system services, and other considerations to minimize and defend the system attack surface"), and design characteristics (e.g. "availability, fault tolerance and resilience, scalability, usability, capacity and performance, testability, portability, and affordability").[31] Decisions regarding specific design principles, security considerations, and design characteristics are themselves influenced by decisions on things like software requirements and architecture choices, as well as realities regarding "necessary enabling systems or services," which may include "software and system platforms, programming languages, design notations and tools for collaboration and design development, design reuse repositories (for product lines, design patterns, and design artifacts), and design standards."[32]

While it is perhaps hoped that the decisions that come before and potentially influence the Design Definition process are "as design-agnostic as possible to allow for maximum flexibility in the design trade space," this is not actually a necessity or always even possible, because at its heart the "process is driven by requirements" and mainly "focuses on compatibility with technologies [e.g., hardware and architectures] and other

---

30 *ISO/IEC/IEEE 12207,* 71.

31 Ibid., 72.

32 Ibid., 72, 71.

design elements and feasibility of implementation and integration."[33] Additionally, in light of the fact that issues may arise during the implementation and integration phases, it is argued that a software design need not be viewed as static once selected, nor must it be fully defined before beginning other phases of development; rather, the Design Definition process "is typically applied iteratively and incrementally to develop a detailed design" and "is usually concurrent with software implementation, integration, verification, and validation."[34]

This approach to defining a piece of software's design—"which accommodates flexibility in design while the software is being constructed"—fits perfectly with the Agile development approach, in which "software design, implementation (construction), and continuous integration are frequently performed concurrently."[35] The Agile approach is very popular in industry, "because it is believed to be more affordable and to deliver usable products more quickly."[36] This is mainly due to the fact that "concept exploration, development, construction, testing, transition, and retirement of previous software can be performed concurrently for successive iterations," potentially reducing or eliminating the time that development teams must wait for others to complete their portions of a project, therefore possibly reducing the overall time it takes to fully develop and deploy a piece of software.[37] This way of doing things "is contrasted with a formal top-down approach… in which construction cannot begin until design is approved so that the constructed software [can be] traced to a previously approved detailed design"—that is to say, something more along the lines of a linear "sequentially staged (idealized waterfall)" approach to software development.[38]

In the end, while the Design Definition process does not necessarily preclude the use of an extensible-minded design pattern such as MVC, it seems to not particularly

---

[33] Ibid., 71.

[34] Ibid.

[35] Ibid., 127

[36] Ibid.

[37] Ibid.

[38] Ibid.

encourage it either. In many ways, the process itself and whatever design is initially or eventually utilized are orthogonal. What is encouraged by the process is continual "replanning," as an application's design or design pattern is changed to accommodate ad hoc solutions to issues encountered throughout development.[39] Working within this type of development environment, it is not hard to imagine the strict adherence to an extensible-minded design pattern as being potentially difficult and problematic, and therefore probably not the best way to attempt to ensure extensibility. This highlights the second motivating force behind this thesis. In many ways, in order to encourage the development of extensible code, what is needed is probably something that is a bit more flexible than a design pattern: perhaps, something along the lines of a distillation of the key features of extensible-mined design patterns such as MVC, which could be used as an educational guide and with greater ease within the context of the Design Definition process.

[39] Ibid., 127, 71.

III. MVC OVERVIEW AND ANALYSIS

A. MVC DESIGN PATTERN BACKGROUND

The Model-View-Controller design pattern is probably one of the more logical design patterns to investigate in terms of extensibility, simply because it is time-tested and widely used by industry leaders. Tryve Reenskaug is credited with first developing MVC for the Smalltalk-80 programming language at Xerox PARC (Palo Alto Research Center) in 1978.[40] Working within the context of the object oriented programming paradigm, the pattern's "top-level goal was to support the user's mental model of the relevant information space and to enable the user to inspect and edit this information."[41] This meant thinking about the user's mental model as something that was both "accessible through the user interface," but also—given the inherent differences between users in terms of their thinking, perspectives, and desires, as well as the wide availability of different types of interfaces—not necessary correspondent with any actual raw data formats, program objects, or domain services found within the system.[42] Thus, at its most basic level, one of the design pattern's chief concerns was how to systematically construct flexible and adaptable software that was able to "create the illusion that the system implements the user's mental models."[43]

Reenskaug describes this illusion as the "direct manipulation metaphor: the sense that end users are actually manipulating objects in memory that reflect the images in their head."[44] He explains the way in which the Model-View-Controller design pattern supports the metaphor in the following manner:

---

[40] Tryve Reenskaug, "The Model-View-Controller (MVC): Its Past and Present," University of Oslo, 6, last modified August 20, 2003, https://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf.

[41] Ibid. 1.

[42] Ibid., 8.

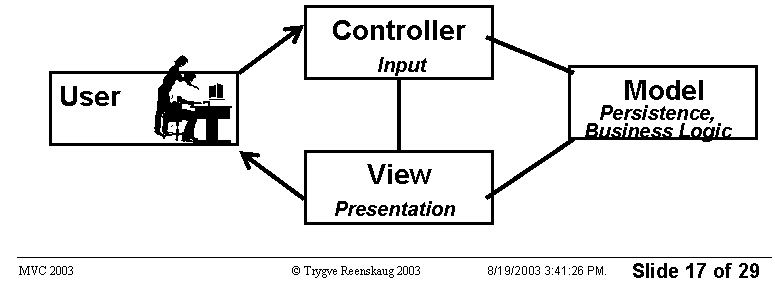
[43] Ibid., 9.

[44] Trygve Reenskaug and James O. Coplien, "The DCI Architecture: A New Vision of Object-Oriented Programming," Vilnius University, 2, last modified March 9, 2009, https://klevas.mif.vu.lt/~donatas/Vadovavimas/Temos/DCI/2009%20The%20DCI%20Architecture%20-%20A%20New%20Vision%20of%20OOP.pdf.

> The job of the *model* is to "filter" the raw data so the programmer can think about them in terms of simple cognitive models. For example, a telephone system may have underlying objects that represent the basic building blocks of local telephony called *half-calls*. (Think about it: if you just had "calls," then where would the "call" object live if you were making a call between two call centers in two different cities? The concept of a "half-call" solves this problem.) However, a telephone operator thinks of a "call" as a thing, which has a duration and may grow or shrink in the number of parties connected to it over its lifetime. The Model supports this illusion. Through the computer interface the end user feels as though they are directly manipulating a real thing in the system called a "Call." Other Models may present the same data (of a half-call) in another way, to serve a completely different end user perspective. This illusion of direct manipulation lies at the heart of the object perspective of what computers are and how they serve people.
>
> The View displays the Model on the screen. View provides a simple protocol to pass information to and from the Model. The heart of a View object presents the Model data in one particular way that is of interest to the end user. Different views may support the same data, i.e., the same Models, in completely different ways. The classic example is that one View may show data as a bar graph while another shows the same data as a pie chart.
>
> The Controller creates Views and coordinates Views and Models. It usually takes on the role of interpreting input user gestures, which it receives as keystrokes, locater device data, and other events.
>
> Together, these three *roles* define interactions between the objects that play them—all with the goal of sustaining the illusion that the computer memory is an extension of the end user memory: that computer data reflect that end user cognitive model.[45]

The interaction envisioned by Reenskaug between a user and the Model, View, and Controller of an application is diagrammed in Figure 1.

[45] Ibid, 3.



An illustration of Reenskaug's vision of interaction between the user and the various components of a MVC application, with some of the interactions being unidirectional and others being bidirectional.

Figure 1. Tryve Reenskaug's vision of MVC[46]

Over the years, the MVC design pattern has become widely adopted by industry leaders, who argue its many benefits. Apple has argued that "MVC is central to a good design for a Cocoa application."[47] This is in part due to the fact that "many Cocoa technologies and architectures are based on MVC and require that… custom objects play one of the MVC roles."[48] But along with this argument for consistency, it is also stated that MVC applications have numerous benefits over non-MVC applications, including the fact that "many objects in these applications tend to be more reusable," "their interfaces tend to be better defined," and they "are also more easily extensible than other applications."[49] Microsoft similarly argues the benefits of MVC, stating that utilization of the Model, View, and Controller roles within an application generally helps developers "scale the application in terms of complexity because it's easier to code, debug, and

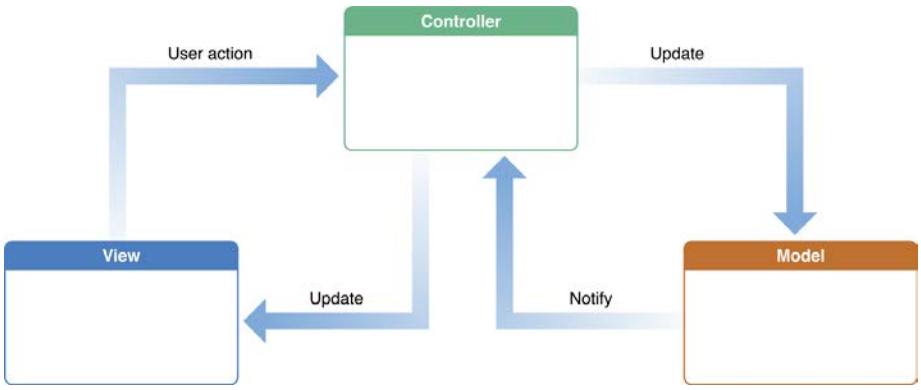
[46] Source: Reenskaug, "The Model-View-Controller (MVC): Its Past and Present," 10.

[47] Apple, "Model-View-Controller."

[48] Ibid.

[49] Ibid.

test."[50] Figures 2 and 3 illustrate Apple's and Microsoft's vison of the MVC design pattern, which are fairly similar to Reenskaug's, perhaps most differing from the original vision in that they both add the concept of directionality to all interactions between modules.

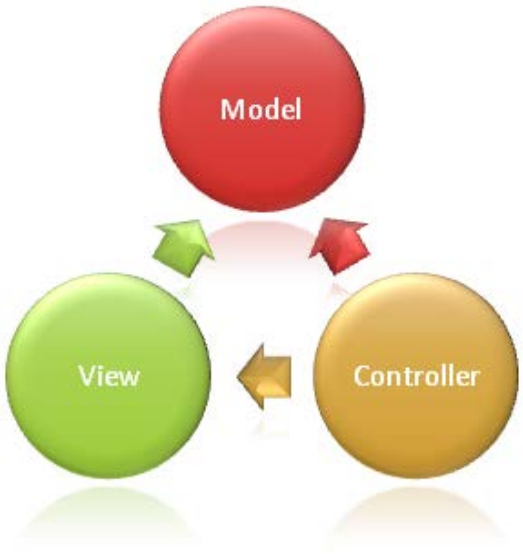


Apple's vision of the interactions between the components of a MVC application. While the general structure is the same as that seen in Figure 1, there are notable differences between this and Reenskaug's vision of MVC, included the unidirectional nature of all interactions and the lack of direct interaction between the View and Model components.

Figure 2. Apple's vision of MVC[51]

[50] Steve Smith, "Overview of ASP.NET Core MVC," Microsoft, last modified January 7, 2018, https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1.

[51] Source: Apple, "Model-View-Controller."



Microsoft's depiction of the interactions between the components of a MVC application. This third depiction illustrates yet another different view on the directionality of component interactions and potential component hierarchy. The fact that disagreements exists regarding the best way to organize MVC components is perhaps one way to explain the impetus behind some of the MVC-like alternatives described in the next section.

Figure 3. Microsoft's vision of MVC[52]

Of course, this is not to say that the MVC design pattern is without its critics. While it is a popular pattern, so are numerous MVC-like alternatives, each of which seems designed specifically in an attempt to overcome some perceived shortcoming of MVC when it comes to implementing real-world applications.

B. MVC-LIKE ALTERNATIVES

If one searches for guidance on how to implement MVC, one will invariably run into countless papers, articles, blog posts, and courses outlining the issues some developers have experienced in their attempts to develop MVC compliant applications. These issues usually center on one or more ways in which use of the pattern can fail to deliver on its promised benefits, resulting in an application that is not flexible or adaptable, or possibly requires substantial refactoring if its code is to be reused or extended. Additionally, in perhaps an attempt to differentiate and perhaps even brand the

[52] Source: Smith, "Overview of ASP.NET Core MVC."

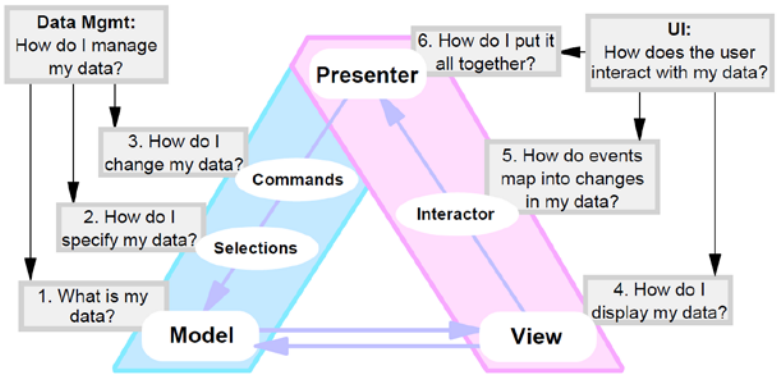
argued solution to the encountered issues, these writings also tend to propose changes to the MVC acronym. The following are some common MVC-like alternatives:

- **Model-View-Presenter (MVP):** This MVC-alternative was developed by IBM for the Taligent programming language in the 1990s in an attempted to enforce "clean separation and encapsulation" between the data that is stored in a Model and the information that is viewed and manipulated by a user, as well as better respond to the fact that, especially with graphical user interface (or GUI) applications, users tend to view and interact with information through the same interface.[53] To meet these goals, the pattern attempted to "formalize the separation between the Model and the View Controller" by defining all displayed and interactable information (i.e., non-Model data) as a distinct application component called a "Presentation."[54] The application's View Controller then had its operations refined and restricted to specifically handling Presentations in a unidirectional manner—receiving updates from Views and sending updates to Models—hence its name change to "Presenter."[55] Figure 4 illustrates the interaction between the Model, View, and Presenter modules of an MVP application.

[53] Mike Potel, "MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java," Taligent, Inc., 1996, last modified September 11, 2014, https://www.researchgate.net/publication/255616200_MVP_Model-View-Presenter_The_Taligent_Programming_Model_for_C_and_Java_Taligent_Inc, 2.

[54] Ibid, 3.

[55] Ibid, 6.



An illustration of the interactions between the components of the MVP design pattern as envisioned by Taligent. Of particular note here is the replacement of the Controller component with the Presenter component, as well as the strict directionality of interactions between the Model, View, and Presenter.

Figure 4. The MVP design pattern.[56]

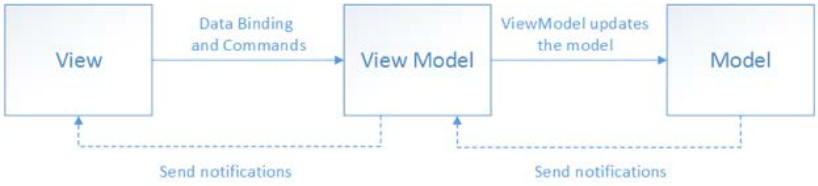
- **Model-View-ViewModel (MVVM):** Microsoft argues that this pattern "helps to more cleanly separate business and presentation logic"; that is to say, just like MVP, MVVM attempts to more formally separate the data that may be used by the system and stored in Models from the information that is utilized in user interfaces (or UIs).[57] But instead of relying on Presentation and Presenter components to achieve this end, MVVM uses a ViewModel component, which acts as an intermediary between an application's Model(s) and View(s), carrying out updates to system data and providing digestible information to UIs.[58] IBM argues that a strong benefit of this pattern is that ViewModels tend to contain code that would otherwise be located in a View Controller or Presenter, thus the use of ViewModels can help reduce or at least better manage the size of those

[56] Source: Ibid, 7.

[57] David Britch and Craig Dunn, "The Model-View-ViewModel Pattern," Microsoft, last modified August 6, 2017, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

[58] Ibid.

components.[59] In this manner, the ViewModel does not replace the Controller as much as assist it. The Controller need only concentrate on transitions between the potentially numerous View-ViewModel-Model pairings found with an application. Figure 5 illustrates the interaction of the Model, View, and ViewModel modules of an MVVM application.



An illustration of the components of the MVVM design pattern. Not shown here are Controller components, which still exist in implementations of MVVM, handling transitions between the different View-ViewModel-Model pairings that may be found within the application. The key point here is simply that the View and Model do not interact directly, but rather utilize a ViewModel as an intermediary, which can house code and logic for how to manipulate information that is sent from the View to be saved as data in the Model, as well as how to manipulate data that is retrieved from the Model so that it can be displayed by the View.

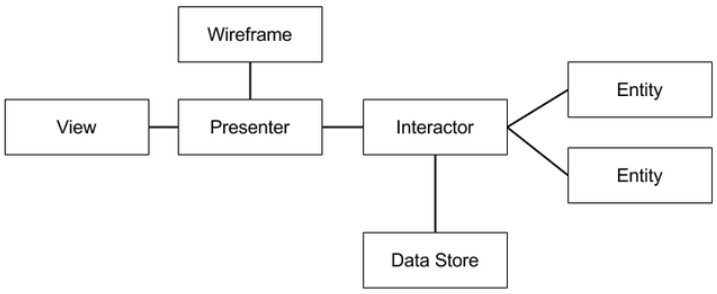
Figure 5. The MVVM design pattern[60]

- **View, Interactor, Presenter, Entity and Router (VIPER):** This pattern takes MVVM's decomposition and creation of new components even further. First, the Model is essentially decomposed into two parts: A Data Store module for persistent data storage and Entity modules for data that is in use. Then, the MVC Controller is replaced with three components: A Presenter module for handling UI setup and updates, an Interactor module for handling interactions with any Data Stores or Entities, and a Router (or Wireframe) module, which handles transitions to and from other application scenes (i.e., other View, Interactor, and Presenter [or VIP]

[59] Taylor Franklin, "Control your View Controllers with a Model-View-ViewModel Architecture," IBM, last modified January 15, 2016, https://developer.ibm.com/open/2016/01/15/control-your-view-controllers-with-a-model-view-viewmodel-architecture/.

[60] Source: Britch, "The Model-View-ViewModel Pattern."

module groups).[61] The argument supporting this perhaps extreme level of decomposition is that it helps avoid the occurrence of what is often jokingly referred to as the "Massive View Controller," which is a View Controller that has become bloated and unmanageable as it incorporates the code for everything from data formatting and UI setup to networking, user input validation, segues between scenes, and more.[62]



An illustration of the interactions between the components of the VIPER design pattern. Not shown here are the potential connections between the components of one VIP module group and another.

Figure 6. The VIPER design pattern[63]

There are many other alternative design patterns, but the basic trend here seems to be clear. For the most part, the patterns all attempt to offer better guidance for developers on how to either more effectively separate MVC components or reduce the size of bloated components by decomposing them into smaller ones with more restricted roles. This points to the fact that patterns such as MVP, MVVM, or VIPER are not really rejections of MVC. At their core, they actually embrace its general philosophy and aesthetics and operate in an essentially identical fashion, just with perhaps slight structural changes, such as two or three components in place of an original one. In a way, they simply

61 "Meet VIPER: Mutual Mobile's application of Clean Architecture for iOS apps," Mutual Mobile, last modified September 24, 2014, https://mutualmobile.com/posts/meet-viper-fast-agile-non-lethal-ios-architecture-framework.

62 Ibid.

63 Ibid.

attempt to help developers write better MVC applications, through the application of certain tweaks and refinements to the MVC pattern's overall structure.

C. MVC KEY FEATURES

By ignoring the differences between MVC and MVC-like alternatives and instead focusing on their similarities, three shared features of the design patterns quickly become apparent.

One feature is the use of modularity. The only disagreement in this area seems to be to what degree it is necessary. Both MVC and MVP argue that three general module types are enough, while MVVM argues that four are needed (seeing as how Controllers/ Presenters are not actually replaced by the inclusion of ViewModels, just possible assisted). VIPER meanwhile argues for six.

A second feature is the need for clear separation between those modules, or decoupling. Here, disagreements seem to center mostly on what level of separation is required and how to implement it. MVP makes the argument that better decoupling can be achieved by just reworking an already present module (i.e., the Controller), redefining its role and strictly limiting how it goes about it. Alternatively, MVVM takes the approach that a new intermediary module is necessary to better separate two modules—Models and Views specifically—which are perhaps prone to tight coupling.

The third feature is a clear-cut, organized structuring of modules, perhaps best described as a hierarchy. Disagreements seem to focus on which modules should actually interact with one another and the direction of those interactions, which in practical terms could easily inform which module acts as the caller of the other. While Reenskaug's original vision of MVC does not seem to concern itself much with the directionality of module interactions, both Apple's and Microsoft's versions of the pattern do. Apple's version of MVC, like the MVVM design pattern, doesn't allow for direct interaction between the Model(s) and View(s) of an application, instead requiring the use of an intermediary module. In both situations, the central position of these intermediary modules makes them likely candidates as root or caller modules. Meanwhile, one of MVP's main distinguishing principles is the fact that the one of its modules—the

Presenter—operates in a highly restricted unidirectional manner, only receiving updates from one module type and only sending updates to another.

Given the fact that these three features of modularity, module decoupling, and module hierarchy are what seem to be shared by all of the MVC and MVC-like design patterns—not specific module names, numbers, roles, or placements—it seems reasonable to assume that they are the contributing factors to the flexibility and adaptability often associated with the patterns and the main reasons for their shared success and popularity among industry leaders. They are the features that are actually put to use by developers in order to make flexible and adaptable software, capable of accommodating a wide variety of users and interface types, and give life to Reenskaug's direct manipulation metaphor. It therefore makes sense to focus on these three features when attempting to develop criteria for extensible software development.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CRITERIA FOR EXTENSIBLE SOFTWARE DEVELOPMENT

Using the three key features identified in the previous section of modularity, module decoupling, and module hierarchy, three criteria can be extrapolated for extensible software development. When applied together, the notion is that they should result in the quality of extensibility—if not in general, then at least in a similar fashion or to a similar degree as relying upon an extensible-minded design pattern such as MVC or one of its alternatives. These are not superficial aspects of an application's code base, like the use of particular component naming conventions, but rather fundamental decisions regarding software structure and operation. Additionally, they are not necessarily independent, but can be seen as interacting with one another, each helping facilitate the next and/or building off the foundation established by the previous.

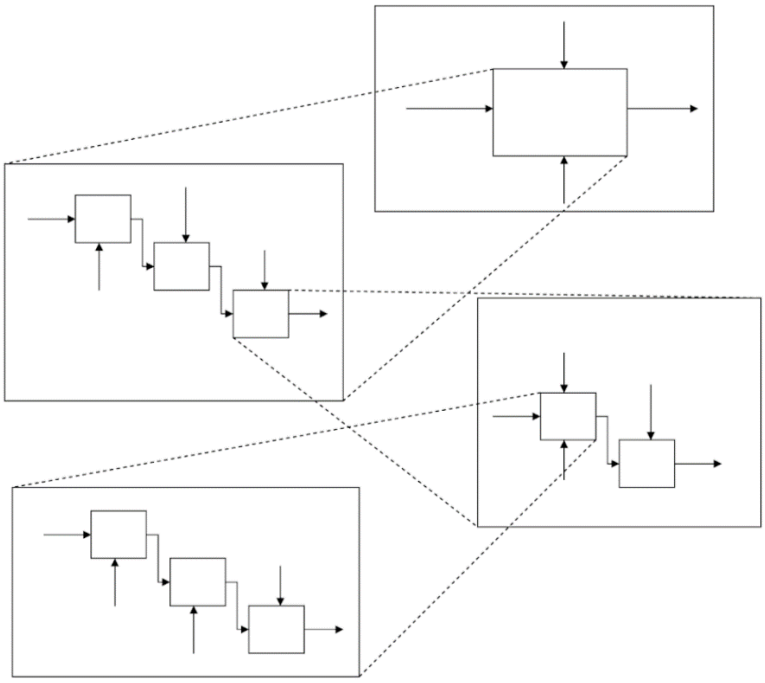
A. MODULE DECOMPOSITION

As a first criterion, with the goal of creating modularity, an application should be decomposed into various distinct modules made up of highly related code. This can be "achieved by grouping together logically related elements, such as statements, procedures, variable declarations, object attributes and so on in increasingly greater levels of detail."[64] Deciding what constitutes a logical relation between different portions of code—and therefore deciding which elements should be grouped together or set apart—can be argued to be somewhat subjective, especially given the fact that all code found within an application should in some way be related. But perhaps a good way to think about relationships is merely in terms of distinct application tasks.

With any task, there is a starting point, ending point, and a metric of success or failure, which can be observed either when the task is finished or along the way. This fact aids in the decomposition process, because it provides a determinable threshold of logical relation: that is to say, any portions of code that perform operations between a task's beginning and end in order to produce some measurable element should probably be

64 Philip A. Laplante, *What Every Engineer Should Know About Software Engineering*, (Boca Rotan, FL: CRC Press, 2007), 85.

grouped together. But even better, thinking in terms of tasks has two distinct benefits. The first benefit is testability. With task-based modules, there is a clear correlation between the structure of the module and how the module and its code can be tested. In the end, the module need only be started, possibly allowed to finish, and the results of its labor compared against some expected value(s). The second benefit of thinking in terms of task-based modules is that they for the most part reveal when further decomposition is probably unnecessary. After all—through the process of separating tasks, then separating those tasks into subtasks, and so on—at some point a task will become essentially irreducible, with each subsequent attempt at decomposition returning the same (or for all intents and purposes, equivalent) start and end points and measurable element(s). Figure 7 illustrates the process of decomposing a module into subordinate modules.



In this illustration, modules are depicted as being decomposed into smaller and smaller modules. "The arrows represent input and outputs in the procedural paradigm. In the object-oriented paradigm, they represent method invocations and messages. The boxes represent encapsulated data and procedures in the procedural paradigm. In the object-oriented paradigm they represent classes."[65]

Figure 7. Module decomposition[66]

B. MODULE DECOUPLING

As a second criterion, with the goal of producing loosely coupled modules, modules should be decoupled to the furthest degree that is possible. In the most general terms, this means that internal design of one module should not be reliant on the internal design of another. This is perhaps best facilitated through the use of "information hiding," which (if done well) results in situations where "only the function of the code [of a particular module] … is visible to other modules, not the method of implementation."[67]

---

[65] Ibid, 86.

[66] Ibid.

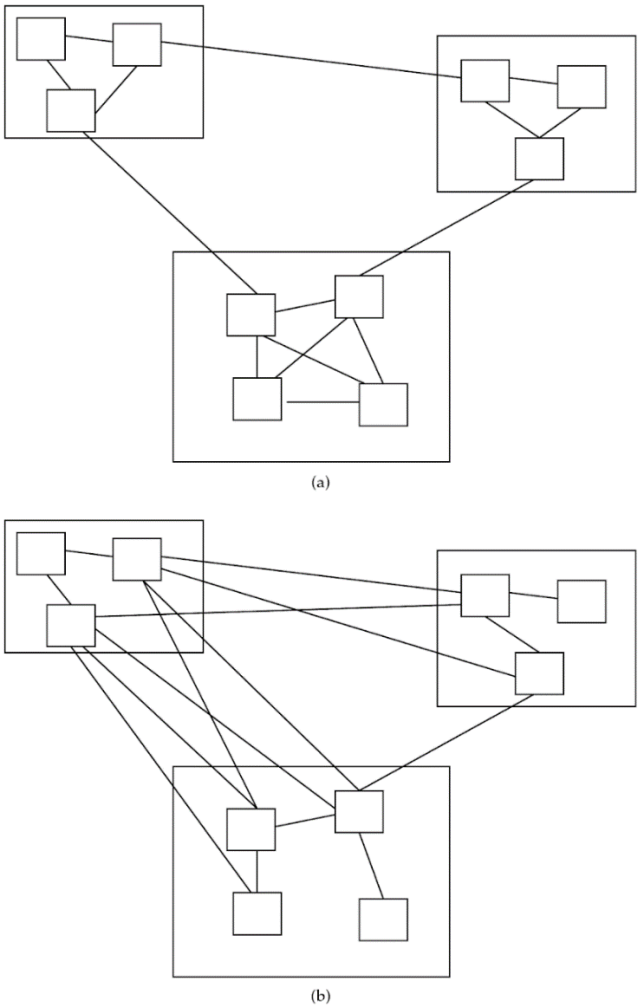
[67] Ibid, 88, 90.

This manner of decoupling modules is effective simply by virtue of the fact that modules cannot become reliant on what they cannot see.

Thinking primarily in terms of objected-oriented languages, information hiding is for the most part achieved through encapsulation and the use of public and private elements. By only allowing each module's data or behavior to be accessed by others "via a published interface," modules cannot directly access, manipulate, or retrieve the data of one another (actions sometimes referred to as "inappropriate intimacy").[68] As a result, they must independently operate on the data they are allowed to retrieve from one another. This may end up in more overall work having to be done on the data itself, but it also reduces the chance that the operations of any two modules will become inseparable.

Of course, even with the use of encapsulation, it is possible for modules to become overly aware of each other's methods of implementation. This can easily occur when modules trade data or data structures that have a highly customized nature or format, which possibly hints at, encourages, or even forces the methods of implementation found within all concerned modules. As a result, encapsulation alone may not be enough to encourage loose coupling and efforts should be made to keep the data and behavior that is accessible through published interfaces as general and implementation agnostic as possible, thus supporting actual independence of implementation. Figure 8 shows two versions of the same theoretical application—one that has been effectively decoupled, resulting in a minimum of interactions, and another that has not been effectively decoupled, resulting in high likelihood of implementation dependencies among its three modules.

[68] Ibid, 94; Martin Fowler, *Refactoring: Improving the Design of Existing Code*, (Boston: Addison Wesley Longman, 1999), 85.



An illustration of (a) loosely coupled or decoupled and (b) tightly coupled modules. "The inside squares represent statements or data, and the arcs indicate functional dependencies."[69]

Figure 8. Module decoupling[70]

C. MODULE HIERARCHY

Finally, as a third criterion, to produce module hierarchy, module interactions should be as one-way as possible, corresponding to clear-cut and easy to follow master/ detail or caller/callee relationships between the components.

---

[69] Laplante, *What Every Engineer Should Know About Software Engineering*, 87.

[70] Ibid.

In programming languages such as Swift and C#, these module interactions are then facilitated through the inheritance of often highly standardized abstract classes, interfaces, or protocols. These structures enforce certain characteristics when it comes to module interactions, not only in terms of what behaviors may be accessed and data can be sent back and forth, but also how. They discourage two-way coupling (i.e., when a module acts as both a caller and callee of another module) and encourage delegation and the presence of clear module levels or tiers within the application's overall structure.

This last point is of particular importance and worth stressing. As David Lorge Parnas explains in his essay "On the criteria to be used in decomposing systems into modules," hierarchical, leveled or tiered organization has two very strong benefits.[71] For one, it promotes greater simplicity of modules, because they can rely on the services of lower priority, higher level, callee, or detail modules to perform work for them, freeing them from the burden of having to implement the services themselves.[72] Second—and perhaps of greater importance in a discussion on extensible software development—a hierarchical organization allows one to "cut off the upper levels and still have a usable and useful product."[73] Parnas goes on to explain:

> The existence of the hierarchical structure assures us that we can "prune" off the upper levels of the tree and start a new tree on the old trunk. If we had designed a system in which the "low level" modules made some use of the "high level" modules we would not have the hierarchy, [therefore] we would find it much harder to remove portions of the system, and "level" would not have much meaning in the system.[74]

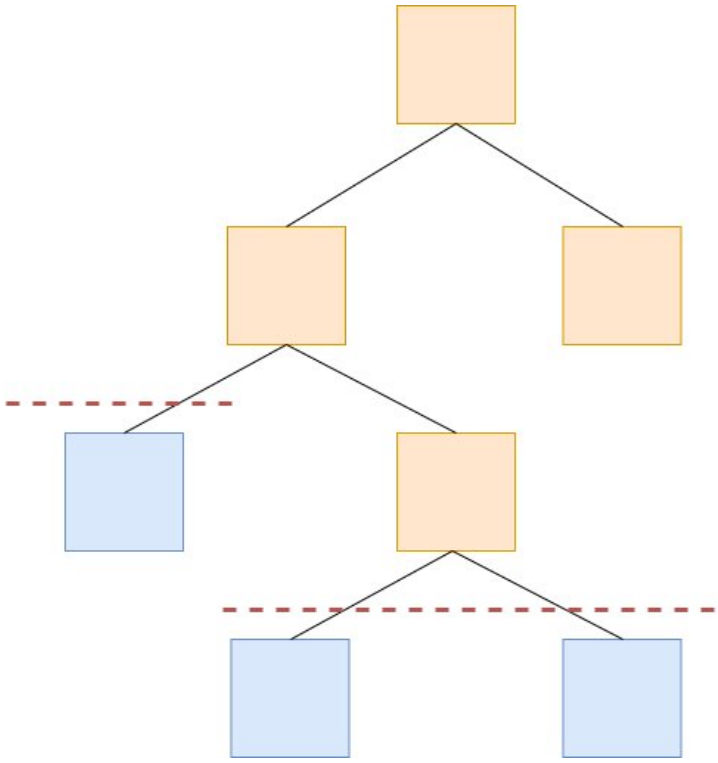
---

[71] David Lorge Parnas, "On the Criteria To Be Used In Decomposing Systems Into Modules," (Pittsburgh, PA: Carnegie Mellon University, 1971), 23, last modified June 30, 2018, https://figshare.com/articles/On_the_criteria_to_be_used_in_decomposing_systems_into_modules/6607958.

[72] Ibid, 23.

[73] Ibid, 23.

[74] Ibid, 32.



In this figure, the modules are structured into a hierarchy, with the solid black lines between them representing interactions. Orange modules are to be retained while blues modules represent those that can be pruned and replaced or modified to extend the application and add new functionality.

Figure 9. Hierarchical module structure

These thoughts are clearly related to the goal of extensible software development. It is perhaps therefore not surprising that a practice known as “Parnas partitioning,” in which developers attempt to push code and functionality that is likely to be changed in the future into higher level and prune-able modules is often argued as a best practice when it comes to code modularization, information hiding, and extensible software development.[75]

D. CRITERIA OVERVIEW

In looking at the criteria outlined earlier for extensible software development, it is easy to see how each can potentially interact with and help facilitate the next. The

[75] Laplante, *What Every Engineer Should Know about Software Engineering*, 90.

decomposition of code into modules can help create the basic building blocks of an application. The code—separated into logically related groupings—is easier to work with, debug, and test. The localization of issues and code becomes easier and the code base is therefore much more modifiable from the perspective of developers and testers. With modules present, it would then be nice to limit the ways in which refactoring one module might require the refactoring of another. This is where module decoupling comes in. With modules that are loosely coupled, the overall code base becomes far more reusable, as modules that are not altered can potentially just be reused. With modules that have been decoupled to a sufficient degree present, the last concern is module hierarchy. As modules interact, performing services for one another, they can become dependent on one another. Having modules that are organized in a way that allows for the easy refactoring, removal, or replacement of certain modules while limiting the negative effects of such actions on the application, including the need to substantial refactor neighboring modules, is really the last required element for software extensibility. Therefore, it makes sense that, when developing or reviewing software for use by the DoD, these three criteria should be looked for and rated. Put another way, if modules are not found to be cleanly decomposed and decoupled, or the organization of an application's code is confusing and prone to things like circular or multilevel dependencies, it is probably reasonable to assume that that application will not be very extensible.

V. CASE STUDY

While the overview of the criteria for extensible software development in the previous chapter may be helpful for some, others may gain a better understanding of the discussed concepts by viewing ways in which they may be applied in a real-world context. What follows is a case study of a simple to-do list application. The application is written in Swift—a modern object-oriented programming language used for Apple devices—and for the most part conforms to the MVC design pattern, with perhaps some slight variations. However, perhaps most importantly, the code of this application demonstrates how the criteria discussed in the previous chapter might be found in real-world, working software. (The code belonging to modules that are mentioned but not directly referenced in this chapter can be found in the appendix of this thesis.)

A. MODULE DECOMPOSITION

As discussed previously, an application should be decomposed into modules. That is to say, it should be organized into distinct groupings of logically related portions of code, often defined by roles or tasks performed by the application. This will help make the code base easier to work with, test, and debug, and thus far more modifiable.

Unfortunately, when developers think of module decomposition, they might not think critically enough. For example, they may think only in terms of Models, Views, and Controllers. In reality, these modules should probably only be thought of as general guides. More to the point, they do not perfectly define what code should or should not be grouped together or separated out. This means that programmers should always think critically about the code they are developing, regardless of what module naming conventions (e.g. MVC, MVVM, MVP) they may be relying upon, always asking themselves if code that is located within a particular module is essential to that module's defined role and tasks. If it is not, and it can be separated out into a distinct module, it probably should be.

An excellent example of this type of code, which can confuse developers in terms of where it should or should not go, can make up a module sometimes known as a Model

Manager. A Model Manager typically works between a Model (or Models) and a View Controller, taking on the role of an intermediary. It receives signals from the View Controller, then gets or sets data stored in a Model accordingly. Sometimes it returns information—which is itself the result of copied and possibly then manipulated data stored in a Model—back to the View Controller. However, perhaps most importantly, the code contained within a Model Manager could easily be found within the View Controller in certain situations.

There are various reasons why one would probably not want to put the code found in a Model Manager into a View Controller. One main reason, discussed in the next section of this chapter, involves raising the likelihood of tight coupling between the Model(s) and View Controller. The second reason has to do with the already mentioned issue of module bloat, which often occurs when Controllers attempt to incorporate too many tasks and their code.

Generally speaking, the less code and fewer tasks contained within a module, the better. Less code means the module is probably easier to read and fully understand, which can significantly aid those developing or maintaining the code base of the application. Fewer tasks means the module will more than likely be easier to test and debug, as it will probably have a lower number of inputs, outputs, and internal operations to check and validate. To avoid Controller bloat, the Master View Controller class found within this thesis's case study application relies upon the use of a Model Manager, which is comprised of the following code:

```
//
//  ElementManager.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 4/8/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import UIKit

import os.log

class ElementManager {

    // MARK: Properties
```

```
private var elementGroup: [String: Element]

// MARK: Initialization

init() {

    elementGroup = [String: Element]()

    loadElements()
}

// MARK: Public Methods

func getElementTemplate() -> [String: String] {

    return [“title”: “Empty,”
            “body”: “Empty,”
            “dueDate”: Date().description]
}

func addElementUsing(template: [String: String]) {

    let newEntry = Element(title: template[“title”]!, body:
        template[“body”]!, dueDate: template[“dueDate”]!)

    self.elementGroup[(newEntry.getCreationDate())] = newEntry

    saveElements()
}

func updateElementUsing(creationDate: String, updates: [String:
    String]) {

    self.elementGroup[creationDate]?.set(title: updates[“title”]!)

    self.elementGroup[creationDate]?.set(body: updates[“body”]!)

    self.elementGroup[creationDate]?.set(dueDate:
        updates[“dueDate”]!)

    saveElements()
}

func removeElementWith(creationDate: String) {

    self.elementGroup[creationDate] = nil

    saveElements()
}

func getContentsOfElementWith(creationDate: String) ->
    [String:String] {

    return [“body”: (self.elementGroup[creationDate]?.getBody())!,
```

```
                "title":
                    (self.elementGroup[creationDate]?.getTitle())!,
                "dueDate":
                    (self.elementGroup[creationDate]?.getDueDate())!]
    }

    func elementCount() -> Int {

        return self.elementGroup.count
    }

    func sortedElementCreationDates() -> [String] {

        return [String](elementGroup.keys).sorted()
    }

    // MARK: Private Methods

    private func saveElements() {

        let isSuccessfulSave =
            NSKeyedArchiver.archiveRootObject(elementGroup, toFile:
                Element.ArchiveURL.path)

        if isSuccessfulSave {

            os_log("Elements successfully saved.," log:
                OSLog.default, type: .debug)

        } else {

            os_log("Failed to save items...," log: OSLog.default, type:
                .error)
        }
    }

    private func loadElements() {

        if let savedElements =
            NSKeyedUnarchiver.unarchiveObject(withFile:
                Element.ArchiveURL.path) as? [String: Element] {

            self.elementGroup = savedElements
        }
    }
}
```

One thing that is immediately obvious when viewing the code of the Element Manager class is just how much logic it contains and thus removes from the Master View Controller class. In the case of this to-do list application, if a Model Manger were not used to handle interactions between the View Controller and the elements comprising the Model, the Master View Controller class would have to contain its own elementGroup

property, as well as code that delivered the same functionality as the seven public methods found within the Element Manager class. Additionally, the Master View Controller class would have to incorporate the functionality of the two private methods found within the Element Manager class, meaning the already much larger View Controller would have to also take on tasks that, in the application's current form, the View Controller is not directly concerned with or even aware of.

When one thinks of the numerous tasks that might be handled by a View Controller—including but not limited to interacting with numerous Views and Models, routing aspects of application control and state between other Controllers, and overseeing actions such as hardware communications and networking—the incredible speed and ease in which the module can become bloated is made apparent. That is why, if it becomes possible to separate out the code of any of these tasks and place it in a distinct subordinate helper module, it should probably be done, regardless of whether the application already has its requisite Models, Views, and Controllers.

In this way, looking for an acceptable level of module decomposition can be seen as going beyond simply requiring that an application be composed of modules of particular names and roles. Rather, it is the position that every module should be well defined in its role and limited in the number of tasks it takes on, and that existing modules may need to be broken up and new ones created if this is not the case.

B. MODULE DECOUPLING

One of the reasons module decomposition may not be done is simply because a programmer might view it as too difficult or time-consuming to do. After all, the work often requires planning, because inter-module communications can be more restrictive than the communications between the elements found within a single module. In order to bypass that restrictiveness and required planning, yet still separate code into distinct modules, a programmer might relax their use of information hiding techniques when developing their modules; for example, they may weaken module encapsulation. The result of this can be situations in which the internal elements of various modules interact freely and extensively. While the code of these modules may appear to be separated into

distinct groupings, those separations are essentially nominal, there benefit undone by interactions that for the most part fuse the modules together. This is the essence of the previously mentioned situations of inappropriate intimacy or tight coupling between modules.

Module decoupling focuses on creating loose coupling between modules. Interactions are kept as general as possible and to a minimum. While the function of each module can and probably should be obvious, what goes on inside the module should not be. By not being allowed to know too much about each other's internal code, modules cannot become dependent on their neighbors' implementations. This state of unawareness protects each module from changes that may be done to the others, and vice versa, which in turn raises the likelihood that modules will remain reusable in the face of any potential modifications to the application's code.

As stated in the previous section of this chapter, not using a Model Manager like the Element Manager class could raise the likelihood of tight coupling between the Master View Controller class and the Model(s) it relies upon. This is due to the fact that a Model Manager—in its role as an intermediary between the Model and View Controller—adds a level of abstraction between the two module types. In looking at the code of the Element Manager class, it becomes apparent that the module provides the Master View Controller very little information about the Model(s) being accessed. The module provides a method (named sortedElementCreationDates) that the Master View Controller class can use to get the unique identifiers of all the data elements that are available to it. Those identifiers must then be used to get or set any data elements. How the Element Manager class stores, retrieves, or manipulates the data elements is for the most part a total mystery to the Master View Controller class.

If a Model Manager were not used and the logic that is currently present in the Element Manager class were instead located in the Master View Controller class, the Master View Controller class would contain something along the lines of the elementGroup property. Although currently the property is a dictionary data structure, if located in the Master View Controller, it could easily be refactored as a list data structure. This change might make sense to a developer who wanted to simplify and reduce code,

because the sortedElementCreationDates method could be eliminated. The data stored within the list version of elementGroup could simply be searched for, retrieved, and set using indices. Given the fact that the Master View Controller class inherits iOS's standard Table View Controller class—which utilizes indices rather extensively in its methods for manipulating subordinate Views—the ability to search for, get, and set Model elements through the use of indices would probably be considered far more straightforward, easy, and efficient than using unique identifiers. Additionally, with direct access to the elementGroup property, the Master View Controller class would also have direct access to the Element class objects making up the list. This would allow for the elimination of the addElementUsing, updateElementUsing, removeElementWith, and getContentsOfElementWith methods. String values retrieved from subordinate Views could simply be entered directly into Element class objects, and vice versa. But to do all of this would create a problem. While the need for an entire class and much of its code would be done away with (i.e., the Element Manger class and many of its public methods), such changes would also fuse together the Master View Controller class's methods for updating Views with a very particular implementation of the application's Model component (i.e. Element class objects stored in a list). In short, the View Controller and Model would become tightly coupled and any changes to one module's implementation would probably necessitate extensive refactoring of the other.

In the above scenario, the level of abstraction that separated the View Controller and the Model(s) was allowed to be significantly reduced with the removal of the intermediary Model Manager module. This fact shows how published interfaces like the public methods of the Element Manager class can not only offer access to data and behaviors, but can also help restrict it as well. After all, instead of accepting indices, they require strings in the form of provided unique identifiers. Instead of returning the actual Element class objects stored in the elementGroup property, they construct and return dictionaries. Working in this manner, the methods allow access to Model data, without acknowledging how it is actually stored. When viewed in this manner, it becomes clear that the question one should ask when designing or reviewing an interface is not simply how effectively does it provide information, but also how effectively does it hide

information, abstracting away and not leaking possible clues regarding module implementation.

Taking these efforts a step further, if a caller module has no need to access numerous callee module properties or methods—for example, only requiring a single return message, which might be data or simply an acknowledgement that some task has been successfully completed—then a delegate can be used instead of an interface.

In this thesis's case study application, the Master View Controller class and Detail View Controller class use a delegate to commutate called the Data Delegate protocol. The code of the all three classes looks as follows:

```
//
//  DataDelegate.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 5/24/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import Foundation

protocol DataDelegate: AnyObject {

    func retrieve(objects: [String:String])
}

//
//  MasterViewController.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 4/8/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import UIKit

class MasterViewController: UITableViewController, DataDelegate {

    // MARK: - Properties

    private var elementManager = ElementManager()

    // MARK: - Setup methods

    override func viewDidLoad() {

        print("In MasterViewController.viewDidLoad")
```

```
    super.viewDidLoad()

    self.navigationItem.leftBarButtonItem = self.editButtonItem
}

override func didReceiveMemoryWarning() {

    super.didReceiveMemoryWarning()
}

override func numberOfSections(in tableView: UITableView) -> Int {

    print("In MasterViewController.numberOfSections")

    return 1
}

override func tableView(_ tableView: UITableView,

    numberOfRowsInSection section: Int) -> Int {

    print("In MasterViewController.numberOfRowsInSections")

    return elementManager.elementCount()
}

override func tableView(_ tableView: UITableView, cellForRowAt
    indexPath: IndexPath) -> UITableViewCell {

    print("In MasterViewController.tableView - Creating cell:,"
        indexPath.row)

    let cellIdentifier = "CustomTableViewCell"

    guard let cell = tableView.dequeueReusableCell(withIdentifier:
        cellIdentifier, for: indexPath) as? CustomTableViewCell
            else {

        fatalError("The dequeued cell is not an instance of
            CustomTableViewCell.")
    }
    cell.cellId =
        elementManager.sortedElementCreationDates()[indexPath.row]

    let data =
        elementManager.getContentsOfElementWith(creationDate:
            cell.cellId)

    cell.cellLabel.text = data["title"]

    print("\t Created ," cell.cellLabel.text!, cell.cellId)

    return cell
}
```

```
override func tableView(_ tableView: UITableView, canEditRowAt
    indexPath: IndexPath) -> Bool {

    return true
}

override func tableView(_ tableView: UITableView, commit
    editingStyle: UITableViewCellEditingStyle, forRowAt indexPath:
        IndexPath) {

    print("In MasterViewController.tableView - Deleting cell:,"
        indexPath.row)

    if editingStyle == .delete {

        if let cell = tableView.cellForRow(at: indexPath) as?
            CustomTableViewCell {

            print("\t Deleting ," cell.cellId)

            elementManager.removeElementWith(creationDate:
                cell.cellId)

            tableView.deleteRows(at: [indexPath], with: .fade)
        }
    }
}

// MARK: - Navigation methods

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    print("In MasterViewController.prepareForSegue")

    if let addItemVC = segue.destination as? DetailViewController {
        addItemVC.dataDelegate = self

        if let indexPath = tableView.indexPathForSelectedRow {

            print("\t Cell number," indexPath.row, "was clicked" )

            let cell = tableView.cellForRow(at: indexPath) as?
                CustomTableViewCell

            print("\t Cell ID is ," (cell?.cellId)!)

            let data =
                elementManager.getContentsOfElementWith(
                    creationDate: (cell?.cellId)!)

            addItemVC.set(details: data)
        }
        else {
            print("\t + was clicked" )
```

```
                let data = elementManager.getElementTemplate()

                addItemVC.set(details: data)
            }
        }
    }

    // MARK: - Delegate methods

    func retrieve(objects: [String:String]) {

        print("In MasterViewController.retrieveItem")

        if let indexPath = tableView.indexPathForSelectedRow {

            print("\t Cell number," indexPath.row, "was clicked" )

            let cell = tableView.cellForRow(at: indexPath) as?
                CustomTableViewCell

            elementManager.updateElementUsing(creationDate:
                (cell?.cellId)!, updates: objects)

            tableView.reloadRows(at: [indexPath], with: .automatic)

            print(elementManager.sortedElementCreationDates())
        }
        else {

            print("\t + was clicked" )

            let newIndexPath = IndexPath(row:
                elementManager.elementCount(), section:0)

            elementManager.addElementUsing(template: objects)

            tableView.insertRows(at: [newIndexPath], with: .automatic)

            print(elementManager.sortedElementCreationDates())
        }
    }
}

//
//  DetailViewController.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 4/8/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import UIKit

import os.log
```

```
class DetailViewController: UIViewController {
    // MARK: - Properties
    @IBOutlet weak var saveButton: UIBarButtonItem!
    @IBOutlet weak private var scrollView: UIScrollView!
    @IBOutlet weak private var stackView: UIStackView!
    weak var dataDelegate: DataDelegate?
    private var details: [String: String] = [:]
    // MARK: - Set up methods
    func set(details: [String:String]) {
        print("In DetailViewController.setDetails")
        self.details = details
    }
    override func viewDidLoad() {
        print("In DetailViewController.viewDidLoad")
        super.viewDidLoad()
        let insets = UIEdgeInsetsMake(20.0, 0.0, 0.0, 0.0)
        scrollView.contentInset = insets
        scrollView.scrollIndicatorInsets = insets
        dynamicSetUp()
    }
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
    private func dynamicSetUp() {
        print("In DetailViewController.dynamicSetUp")
        stackView.spacing = 10
        stackView.distribution = .fillProportionally
        let detailKeys = Array(details.keys).sorted().reversed()
        for key in detailKeys {
```

```
            let label = UILabel()

            label.text = key

            stackView.addArrangedSubview(label)

            let textField = UITextField()

            textField.text = details[key]

            textField.backgroundColor = UIColor.white

            stackView.addArrangedSubview(textField)
        }
    }

    private func returnMessage() {

        print("In DetailViewController.returnMessage")

        let labels = stackView.arrangedSubviews.filter{$0 is UILabel}
            as? [UILabel]

        let textFields = stackView.arrangedSubviews.filter{$0 is
            UITextField} as? [UITextField]

        for i in 0..<stackView.arrangedSubviews.count/2 {

            let key = labels![i].text

            let value = textFields![i].text
            details[key!] = value
        }

        print(details)

        dataDelegate?.retrieve(objects: details)

        navigationController?.popViewController(animated: true)
    }

    // MARK: - Actions

    @IBAction func saveButtonClicked(_ sender: UIBarButtonItem) {

        print("In DetailViewController.saveClicked")

        returnMessage()
    }

    // MARK: - Navigation

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

        print("In DetailViewController.prepareForSegue")
```

```
    }
}
```

In observing the code of the Master View Controller and Detail View Controller classes, it quickly becomes clear how little the two actually know about each other, with all of their interactions going through the Data Delegate protocol. By invoking the protocol, the Master View Controller class simply makes it known to other modules that it implements the method (named retrieve) prototyped in the Data Delegate protocol. A callee module of the Master View Controller class does not know anything else about its caller. Working in the opposite direction, when the Master View Controller class assigns a delegate, it need only know that the callee class possesses a Data Delegate type property. The module does not need to know anything else about its callee module, which in this case is the Detail View Controller class. The result of all of this is coordinated interaction between the two classes, but also almost compete unawareness and a lack of concern regarding what the other class is doing with the data or messages it is being sent.

In the end, module decoupling is all about hiding information. Modules should have an ability to interact with one another, but that interaction should be in a way that allows modules to remain as independent as possible, without leaking knowledge of their implementations. Usually this will involve keeping module communications as minimal, but also as general, as possible. If that is kept in mind, neighboring modules cannot be developed or changed in ways that become dependent on neighboring modules' implementations and as a result modules will likely remain more reusable as the application receives changes to its code base.

C. MODULE HIERARCHY

With module decomposition and decoupling both performed, a possible last hurdle to extensibility can be the organization of the modules that make up an application. Modules that work together can quickly form dependencies on one another, in so far as one can come to rely on the other for the completion of critical tasks. While certain dependencies are unavoidable, those that are circular, convoluted, or multilevel in nature can result in code that is difficult to understand, debug, and modify and should

therefore be avoided. Additionally, modules that are highly likely to receive extensive refactoring or replacement should probably be pushed to the branches or leaves of any module hierarchy, where their alteration or removal will directly impact fewer modules found within the application. This means that, when it comes to module organization, whenever possible modules should work together in a simple to follow, caller/callee or master/detail, hierarchical fashion.

Some excellent examples of this kind of clear cut, hierarchical organization are displayed in the Master View Controller class's interactions with both the Element Manager and Detail View Controller classes. In the former case, the Master View Controller class acts as the caller, interacting with the Element Manager through that class's published interface. In the latter case, the Master View Controller class again acts as the caller, but this time interacting with the Detail View Controller class through the use of the Data Delegate protocol. In both of these cases, the relationship is one-way, with the Master View Controller utilizing the other classes for particular subtasks. Neither the Element Manager nor the Detail View Controller (nor any of their subordinate objects) act as the caller of the Master View Controller class.

Given the application's nature as a to-do list, it makes sense that the Master View Controller class is located at the root of these interactions. The Master View Controller class inherits the iOS Table View Controller class, meaning it is specifically setup to manage a vertically scrolling, updatable, cell-based, list-like graphical user interface. This essentially means the Master View Controller class delivers the core functionality most users would probably expect to see in a phone-based to-do list application. This makes it highly unlikely that it will be replaced during the lifetime of the application. On the other hand, if the Model(s) used by the application for its data storage changed (for example, to something like a remote SQL database), the Element Manager class might need to be refactored. Additionally, the Detail View Controller class could easily be replaced with a different class that delivers some new form of functionality in terms of data display and editing. Really, the only real requirements of that new class would be that it inherited the standard View Controller class and implemented the Data Delegate type property.

In the end, the interactions between the Master View Controller class and the Element Manager or Detail View Controller classes demonstrate something very similar to the concept of Parnas Partitioning. They are strictly one-way relationships, in which functionality that is likely to change has been pushed out into easier to replace branch/ leaf modules. Additionally, they show how once this sort of module organization has been achieved, concepts such as inheritance and subclassing can be leveraged to make the replacement of those branch/leaf modules quicker and easier, as the interfaces or delegates through which they communicate are already established. In this way, it seems apparent how a clear-cut caller/callee or master/detail module hierarchy may be the final piece in creating an extensible application.

VI. CONCLUSION

Compared to non-extensible applications, extensible applications can often be built faster, have fewer bugs, are more secure, and have the capability to adapt to users changing needs. Given the DoD's desire to quickly and easily field cost effective, stable, and secure applications that provide a maximum operational advantage, it would seem that the DoD has an interest in developing extensible applications. Unfortunately, the software development documents and processes utilized by the DoD do not seem to fully support or encourage extensible software development. The intent of this thesis was to help create an educational guide that could work within the context of the highly flexible Design Definition process, providing criteria useful in the development or acquisition of extensible software, without requiring strict adherence to any particular extensible-minded design pattern.

Looking to the MVC and MVC-like design patterns that are popular among industry leaders for indications of which key features seem most necessary for extensibility, this thesis came up with three criteria. The first criterion was the decomposition of application code into modules with well-defined roles and limited tasks. The second criterion was the decoupling of those modules through the using of information hiding techniques, resulting in loosely coupled modules with internal operations that are as hidden and abstracted away as possible. The third criterion was the organization of those loosely coupled modules into a clear-cut hierarchical structure, which utilizes straightforward caller/callee relationships, as well as locates unlikely to change functionality in root or lower-level modules and likely to change functionality in easily pruned higher-level branch or leaf modules. Together, these three criteria can be leveraged in the development or acquisition of software code bases that are potentially more modifiable, reusable, and extensible.

Looking forward, the three criteria outlined in this thesis could be tested further using other case study applications. For example, work could be done using examples of extensible programs written in the C or Go languages, examining how procedural programming effects the application's alignment with the criteria. Additionally, more

design patterns, including those outside of the MVC family, could be assessed in order to elucidate what they do well or poorly with regard to ensuring extensibility. Such data could be used to potentially enhance the criteria. In the end, all research of this type could enhance the DoD's (and other interested parties') understanding of extensible software development and help add to the knowledge base that is necessary for the effective development and acquisition of extensible applications.

APPENDIX

The code of the Element class and Custom Table View Cell class of the case study application, mentioned in Chapter V, are included in this appendix.

```
//
//  Element.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 4/9/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import UIKit

import os.log

class Element: NSObject, NSCoding {

    //MARK: Properties

    private var title: String

    private var body: String

    private var dueDate: String

    private var creationDate: String

    //private var userMetaData: String

    //MARK: Initialization

    init(title: String, body: String, dueDate: String) {

        self.title = title

        self.body = body

        self.dueDate = dueDate

        let date = Date()

        self.creationDate = date.description

        //self.userMetaData = ""
    }

    // MARK: Public methods

    func set(title: String) {
```

```
    self.title = title
}

func set(body: String) {

    self.body = body
}

func set(dueDate: String) {

    self.dueDate = dueDate
}
/*
func set(userMetaData: String) {

    self.userMetaData = userMetaData
}
*/
func getTitle() -> String {

    return self.title
}

func getBody() -> String {

    return self.body
}

func getDueDate() -> String {

    return self.dueDate
}
/*
func getUserMetaData() -> String {

    return self.userMetaData
}
*/
func getCreationDate() -> String {

    return self.creationDate
}

// MARK: Private methods

private func set(creationDate: String) {

    self.creationDate = creationDate
}

//MARK: Types

struct PropertyKey {

    static let title = “title”
```

```
    static let body = “body”

    static let dueDate = “dueDate”

    static let creationDate = “creationDate”

    //static let userMetaData = “metaData”
}

// MARK: Archiving Paths

static let DocumentsDirectory = FileManager().urls(for:
    .documentDirectory, in: .userDomainMask).first!

static let ArchiveURL =
    DocumentsDirectory.appendingPathComponent(“elements”)

//MARK: NSCoding

func encode(with aCoder: NSCoder) {

    aCoder.encode(title, forKey: PropertyKey.title)

    aCoder.encode(body, forKey: PropertyKey.body)

    aCoder.encode(dueDate, forKey: PropertyKey.dueDate)

    aCoder.encode(creationDate, forKey: PropertyKey.creationDate)

    //aCoder.encode(userMetaData, forKey: PropertyKey.userMetaData)
}

required convenience init?(coder aDecoder: NSCoder) {

    guard let creationDate = aDecoder.decodeObject(forKey:
        PropertyKey.creationDate) as? String else {

        os_log(“Unable to decode the creation date for an Element
            object.,” log: OSLog.default, type: .debug)

        return nil
    }

    let title = aDecoder.decodeObject(forKey: PropertyKey.title)
        as? String

    let body = aDecoder.decodeObject(forKey: PropertyKey.body) as?
        String

    let dueDate = aDecoder.decodeObject(forKey:
        PropertyKey.dueDate) as? String

    //let userMetaData = aDecoder.decodeObject(forKey:
        PropertyKey.userMetaData) as? String
```

```
        self.init(title: title!, body: body!, dueDate: dueDate!)

        self.set(creationDate: creationDate)

        //self.set(userMetaData: userMetaData!)
    }
}

//
//  CustomTableViewCell.swift
//  toDoListCaseStudyApp
//
//  Created by Damon Alcorn on 4/10/18.
//  Copyright © 2018 Damon Alcorn. All rights reserved.
//

import UIKit

class CustomTableViewCell: UITableViewCell {

    @IBOutlet weak var cellLabel: UILabel!

    var cellId: String = “”

    override func awakeFromNib() {

        super.awakeFromNib()

        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {

        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }
}
```

LIST OF REFERENCES

Apple. “Model-View-Controller.” Developer. Accessed May 22, 2018. https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

Bourque, Pierre, and Richard E. Fairley (eds.). *Guide to the Software Engineering Body of Knowledge*. Version 3.0. Piscataway, NJ: IEEE Computer Society, 2014.

Britch, David, and Craig Dunn. “The Model-View-ViewModel Pattern.” Microsoft. Last modified August 6, 2017. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston: Addison Wesley Longman, 1999.

Franklin, Taylor. “Control your View Controllers with a Model-View-ViewModel Architecture.” IBM. Last modified January 15, 2016. https://developer.ibm.com/open/2016/01/15/control-your-view-controllers-with-a-model-view-viewmodel-architecture/.

International Organization of Standardization. *ISO/IEC/IEEE 12207: Systems and Software Engineering – Software Life Cycle Processes, First Edition 2017–11*. Geneva: ISO/IEC, 2017. Piscataway, NJ: IEEE Computer Society, 2017.

International Organization of Standardization. *ISO/IEC 14767, IEEE Std 14767-2006: Systems and Software Engineering – Software Life Cycle Processes – Maintenance, Second Edition 2006–09-01*. Geneva: ISO, 2006. Piscataway, NJ: IEEE Computer Society.

International Organization of Standardization. *ISO/IEC 15408: Common Criteria for Information Technology Security Evaluation – Part 2: Security Functional Components, Ver. 3.1, Rev. 5*. Geneva: ISO/IEC, 2017.

Laplante, Philip A. *What Every Engineer Should Know About Software Engineering*. Boca Rotan, FL: CRC Press, 2007.

Microsoft. “The MVVM Pattern.” Microsoft Patterns and Practices: Proven Practices for Predictable Work. Last modified February 10, 2012. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10).

Mutual Mobile. “Meet VIPER: Mutual Mobile’s Application of Clean Architecture for iOS Apps.” Last modified September 24, 2014. https://mutualmobile.com/posts/meet-viper-fast-agile-non-lethal-ios-architecture-framework.

National Institute of Standards and Technology. *NIST SP-800-53: Security and Privacy Controls for Federal Information Systems and Organizations, Rev. 4*. Gaithersburg, MD: National Institute of Standards and Technology, 2013.

National Institute of Standards and Technology. *NIST SP-800-600: Systems Security Engineering Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems, Volume 1*. Gaithersburg, MD: National Institute of Standards and Technology, 2018.

Parnas, David Lorge. "On the Criteria to Be Used in Decomposing Systems into Modules." Carnegie Mellon University. 1971. Last modified June 30, 2018, https://figshare.com/articles/On_the_criteria_to_be_used_in_decomposing_systems_into_modules/6607958.

Potel, Mike. "MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java." Taligent, Inc. 1996. Last modified September 11, 2014. https://www.researchgate.net/publication/255616200_MVP_Model-View-Presenter_The_Taligent_Programming_Model_for_C_and_Java_Taligent_Inc.

Reenskaug, Tryve. "The Model-View-Controller (MVC): Its Past and Present." University of Oslo. 2003. Last modified August 20, 2003. https://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf.

Reenskaug, Trygve, and James O. Coplien. "The DCI Architecture: A New Vision of Object-Oriented Programming." Vilnius University. Last modified March 9, 2009. https://klevas.mif.vu.lt/~donatas/Vadovavimas/Temos/DCI/2009%20The%20DCI%20Architecture%20-%20A%20New%20Vision%20of%20OOP.pdf.

Smith, Steve. "Overview of ASP.NET Core MVC." Microsoft. Last modified January 7, 2018. https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California