Designing Uncomplicated Software

Article · October 2018

DOI: 10.26439/interfases2018.n011.2954

CITATIONS
0

READS
31

2 authors:

Michael Dorin
University of St. Thomas
9 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Sergio Montenegro
University Wuerzburg
120 PUBLICATIONS **437** CITATIONS
SEE PROFILE

Recibido: 15 de agosto del 2018
Aceptado: 16 de octubre del 2018
doi: 10.26439/interfases2018.n011.2954

Designing uncomplicated software

Michael Dorin
mike.dorin@stthomas.edu
University of St. Thomas. Minnesota, EE. UU.

Sergio Montenegro
sergio.montenegro@uni-wuerzburg.de
Universität Würzburg. Würzburg, Germany

Abstract

The Agile Manifesto prescribes less focus on tools and processes, and more focus on human interactions. This is a very important and powerful concept; however, many development organizations have interpreted it in terms of no procedures and no processes. This is understandable as many activities, such as the design workflow, are thankless and laborious. When a proper design is missing, the resulting source code may become overly complicated and difficult to maintain. The software design does not have to be arduous as this workflow can be done without pain through an adaptation called Responsibility-Driven Design. This adaptation assigns personalities to the internal components of the software to humanize the operation. The new design workflow is completely compatible with agile concepts such as customer interaction, and produces a credible candidate architecture ultimately resulting in the creation of a less complicated software.

Keywords: ***Agile Manifesto, software design, human interactions, Responsibility-Driven Design***

Resumen

Diseño de *software* no complicado

El Manifiesto Ágil prescribe disminuir el foco en las herramientas y procesos para centrarlo en las interacciones humanas. Este es un concepto muy importante y potente; sin embargo, muchos equipos de desarrollo lo han traducido en términos de no procedimientos y no procesos. Esto es comprensible ya que muchas actividades, entre ellas el flujo de trabajo del diseño, son ingratas y laboriosas. Cuando no se realiza un diseño apropiado, el resultado puede ser un código demasiado complejo y difícil de mantener. El diseño de un *software* no tiene que ser arduo y el flujo de trabajo puede aliviarse con una adaptación denominada "diseño conducido por la responsabilidad" (Responsibility-Driven Design). Esta adaptación asigna personalidades a los componentes internos del *software* para humanizar la tarea. El nuevo flujo de diseño es completamente compatible con los conceptos de agilidad, como la interacción con el cliente, y produce una arquitectura candidata con credibilidad que resultará en la creación de un *software* no complicado.

Palabras clave: ***Manifiesto Ágil, diseño de*** **software**, ***interacciones humanas, diseño conducido por la responsabilidad***

1. Introduction

For whatever reason, designing a software has not been as glamorous as simply writing it. In the days when design relied heavily on flowcharts and data flow diagrams, programmers would complain about management requiring those steps. Some organizations believe architecture design is too expensive and time consuming. Another contributing factor to this, at least at the beginning of a project, is that the software problem to be solved is not well understood (Foote & Yoder, 1997).

Immediately writing a code is seen as a way for engineers to begin understanding the domain with the thought of writing the "real code" later, which more often than not will never happen. In the eyes of the customer and management, the code is working, and the team is demonstrating progress.

At this point, piecemeal growth of the software begins and development starts to grow in an uncontrolled fashion (Foote & Yoder, 1997). Put another way, rather than a design and architecture structuring the code, the code defines the design and architecture. This results in an overly complicated code base which is hard to expand and maintain (Foote & Yoder, 1997).

Many alternatives to this approach have been invented such as the Responsibility-Driven Design. As stated by Rebecca Wirfs-Brock, "Responsibility-Driven Design is a way to design that emphasizes behavioral modeling using objects, responsibilities, and collaborations. In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture." (Wirfs-Brock & Wilkerson, 1989). This concept of Responsibility-Driven Design is beneficial for analyzing and designing workflows of software engineering.

If Responsibility-Driven Design is handled at an even more basic level than as described by Wirfs-Brock, a much unencumbered design workflow is produced which nicely partitions the modules of a system.

The goal of the new workflow is to create a better, less complicated software from a human perspective. As part of this ongoing research, students performed analyses and designs using these techniques, the results of which are provided herein.

2. Background

2.1 Complicated code

When discussing a complicated code, it is important to agree on what the term "complicated code" means. Complicated code is a code which is hard to understand and explain by a human who is reading it. Problems with a complicated code are well known and well documented (Banker, Datar, Kemerer, & Zweig, 1993).

The first step in preventing a complicated code is to reacquaint with what makes software complicated. The paper "Coding for Inspections" described a survey carried out to identify the most basic problems seen by software engineers when reviewing a code. Concern for software complexity is not new as McCabe and Halstead designed complexity measurement metrics more than forty years ago (Dorin, 2018). Though a Google search identifies dozens of newer metrics, Halstead and MCabe are readily available without cost. Software is considered undesirable to review when it has higher cyclomatic complexity and higher Halstead difficulty (Dorin, 2018).

Another aspect identified is how stylistic issues also made source code more complicated to review. Table 1 lists the stylistic violations identified in "Coding for Inspections and Reviews" which most bothered reviewers (Dorin, 2018).

Table 1. *Most unpleasant to review styles*

Style Name
There should be space around operators
Do not write over 120 columns per line
Average length of functions
Indent blocks inside of a function
Put matching braces in same column
Use less than 5 parameters in function
Do not use the question keyword
Avoid deeply nested blocks
Use braces for even one statement

2.2 Irreducible complexity

With the thought of avoiding a complicated code in mind, one might think about irreducible complexity. Irreducible complexity is not a universally accepted concept in the biological sciences. Michael Behe defines irreducible complexity as a single system composed of several well-matched, interacting parts that contribute to the basic function, wherein the removal of any one of the parts causes the system to effectively cease functioning. Put simply, if we take a piece away, the system no longer performs as it was intended to (Behe, 2009). Supporters of intelligent design believe this shows that evolution cannot be completely responsible for life on this planet; there had to be an intelligent creator involved.

The merits of biological intelligent design will not be debated here, but one cannot avoid noticing the parallels between computer/software evolution and biological evolution. A code is said to evolve, but a code cannot evolve without the hand of the creator. Some have suggested a code "rots" if left alone long enough. In practice though we can recognize it is not the code that is rotting, but the environment that it was designed to run on is changing. In computer software, there is no way not to recognize the hand of an arguably intelligent creator. An accounts receivable program may one day evolve into a full accounting system program, but it will not do so by mutation.

As a software engineer, this concept should be kept in mind during analysis and design. If a design is overly complicated, the software engineer should work to eliminate extra complexity, with the final target being reduced until the program can be reduced no more without destroying product functionality. Extraneous parts that do not contribute to the program's functionality should be removed.

3. Components of the new workflow

3.1 Dress rehearsal

Military organizations around the world have used rehearsals for centuries. It is said that the Romans rehearsed battles using sand tables with icons to visualize the battlefield (Smith, 2010). The modern army believes the rehearsal is a tool for commanders to make sure parties involved understand the intent and concept of the operations. Rehearsals provide opportunity to identify inadequacies in a plan that were not previously recognizable. Rehearsals contribute to external and internal coordination. (Army, 2015) In other words, rehearsals of all shapes and sizes are used to ensure efficient battlefield operations. Dress rehearsals can be entire battlefield simulations with whole army units participating, or they can be small, where individuals take on the role of entire units. Events are simulated in real-time and participants act out their responsibility at different points of the exercise (Army, 2015). Obviously, the military is not alone in using rehearsals and this is a powerful tool that can be well used in software engineering.

3.2 Play writing

It may be considered odd that information on writing a play would be included in a discussion of software engineering life cycle models. However, when considering employing rehearsals as tool, using a play as a structure should not be overlooked. In his book, "Writing Your First Play", Roger Hall outlines elements of a play (Hall, 2012). Section 1 covers action and how dynamic action employs verbs. In software engineering, verbs can be used to represent methods or functions in your code.

Section 2 discusses obstacles and conflict, such as the conflict faced by stakeholders who do not have the required software.

Though this technique can work with any architecture design, the Model-View-Controller design pattern (MVC) works very well for this approach. MVC defines a plan for organizing components.

The model portion handles data storing and the algorithms for processing data. The view portion is responsible displaying information and results to the user. The controller is in charge and sends commands to the model and the view (Rosenbloom, 2018).

There are many resources describing how to write a successful play; however, applying artistic information to software design is not always obvious. In playwriting, it is important to come up with a main character, then decide on the conflict or problem (Victor, 2009). Afterwards, it should be decided on a beginning point and show the story in actions and "speech". Don't overdo it: one group of students wrote their play based on Star Wars characters and, upon rereading at a later date, they could not remember the roll of each character.

There is one more suggestion which can be a benefit to the success of a play, especially for new authors. Characters with special skills should be provided or generated before playwriting begins. In the sample play, characters with different skill sets participate in completing the required task.

For example, the "Artista" character is responsible for communication. A "Jefa" character is responsible for the overall operation. Other characters for security, data management, and direct communications are included. See Table 2 for a complete list. The "Profesor" and the "Estudiante" are the only two human characters in the play and they represent the users of the software.

The main character of the play is the "Profesor," though the "Jefa" has an active supporting role. As play writers should give characters a significant problem to solve immediately, in the sample play, the problem to solve is how the professor best communicates with students during class.

3.2.1 Example Play: The happy class

Table 2. *Play characters*

Name	Character Title	Responsibility
Claudia	Oficinista (File clerk)	Stores and retrieves data
Diego	Estudiante (Student)	A student using the system (Human)
Gonzalo	Guachimán (Security)	Provides user validation
Patricio	Profesor (Professor)	A professor using the system (Human)
Rebecca	Jefa (Boss)	Manages software operations
Sergio	Telefonista (Telephone operator)	Provides internal communications
Valeria	Artista (Artist)	Generates output to users

Setting

Three software people (Valeria, Rebecca, Patricio) are sitting around piles of paper showing user stories and use cases. A low-resolution prototype is taped to the wall. Cups are full of coffee. The three are very pleased with the quantity and quality of their requirements gathering analysis but harbor some doubts with respect to moving to the next phase.

Narration

Valeria: This is sure good coffee. Do you think we have enough?

Rebecca: I hope not, I want to wrap up and go home… but now what?

Patricio: Now we have to come up with a candidate architecture, but where do we begin? We have gathered so much information and talked to so many people. We even have fantastic low-resolution user interface prototypes.

Valeria: Perhaps we start small. Rebecca, please grab me a minor use case.

Valeria: Ok, I have the "*Profesor* Logs In" use case, but it is still not obvious how to continue.

Rebecca: I know. Let us pretend to be the software. Perhaps we can get an idea of how to construct this thing!

Patricio: Don't be silly.

Rebecca: Wait, wait, let us just give it a try and see where it takes us.

Patricio: Ok, I will pretend to be the *Profesor* in the use case. Rebecca, you'll be the Software.

Patricio: I'll start.

Patricio: Hola, Rebecca. I want to set up a class.

Rebecca: I am not sure what you want or how to help. Valeria, can you show him what he can do?

Valeria: Ok, I will pretend to be in charge of showing stuff.

Valeria: Ok, here are your options. (Valeria shows Patricio a sheet of paper. Patricio pretends his options are written on it.)

Patricio: I think this is getting closer, but rather than calling you by name, I am going to call you by a title to help this stay organized. I will be the Profesor. Rebecca, you seem to be the boss so I will call you "*Jefa*." Valeria, you seem to be a communicator, so I will call you "*Artista*".

Profesor: Hola, Jefa. I want to set up a class.

Jefa: Artista, please show this *Profesor* his options.

Artista: Profesor, welcome, here is our main screen. Professors need to sign in.

Jefa: STOP! I don't know how people authenticate. I just know how to boss people around. We need somebody like a '*guachimán*' to handle this. (Just then they notice Gonzalo is sitting in the corner.)

Patricio: Hey, Gonzalo, come here for a second. We need you to be a *guachimán* in our software world.

Gonzalo: Hey, a *guachimán*? Wow! That sounds like fun!

Patricio: Ok, let us continue again. Remember: Rebecca is the Jefa.

Profesor: Hey, *Jefa*, I want to set up a class.

Jefa: *Artista*, please show this *Profesor* his options.

Artista: *Profesor*, welcome, here is our main screen: Professors need to sign in.

Profesor: *Guachimán*, here is my info.

Guachimán: *Jefa*, the *Profesor* has signed in.

Jefa: *Artista*, please show the *Profesor* how to create a class.

Artista: *Profesor*, please provide a class name and let us know when you are ready to start.

Profesor: *Jefa*, I want to start a class named SEIS610. It is my favorite class and a fantastic professor teaches it.

Rebecca: STOP. I don't know how to create a class. We need somebody to keep track of all of this. Wait, Claudia, come here. Can you pretend to be an *Oficinista* for us? Please, just for a bit. Ok, Claudia, when I call out "*Oficinista*", you answer.

Claudia: Do I have to call you "*Jefa*"?

Rebecca: Yes, you do.

Rebecca: Ok, now let us continue

Jefa: *Oficinista*, please create a class named SEIS610.

Oficinista: Ok, *Jefa*, here is that class you wanted.

Rebecca: Stop again. Ok, Claudia has created a class for us but how do I talk to it. Claudia does not know anything about talking to classes. It's like we need a telephone operator. Sergio, come here for a second. We need you to pretend to be a telephone operator.

Sergio: Do I have to?

Rebecca: Yes, you do; and yes, you need to call me "*Jefa*".

Rebecca: Alright, let us continue again.

Jefa: *Telefonista*, can you please give me a new channel?

Telefonista: Yes, *Jefa*, here you go.

Jefa: Oficinista, hey, sorry to bug you again, but can you store this channel information?

Oficinista: *Jefa*, consider it done!

Jefa: *Artista*, can you show the *Profesor* a classroom based on the new class with this ID and channel information?

Artista: *Profesor*, here is your class. Make sure you tell your students the ID is 1234! (Diego walks by… he decides to be funny and pretends to be a student.)

Diego: Hey, look at me. I am an *Estudiante*!

Estudiante: Hola, *Jefa*, I want to participate in class with ID 1234.

Jefa: *Oficinista*, do we have a class with ID 1234? If so, can you give it to me?

Oficinista: Yes, *Jefa*, we do have that class; here is the info.

Jefa: *Artista*, can you display a class to this *Estudiante* with this info?

Artista: *Estudiante*, here is your class.

Estudiante: *Jefa*, can you tell the Profesor that I don't understand the problem just described?

Jefa: *Oficinista*, what is the channel that class 1234 uses?

Oficinista: Class 1234 uses this channel.

Jefa: *Telefonista*, can you relay this question to this channel to the *Profesor*?

Telefonista: Yes, I will, and I have.

Patricio: Well done, gang! I think this gives us a fascinating picture! The end!

3.3 Responsibility-Driven Design

Data flow diagrams remain a very popular form of analyzing a system. Data flow diagrams, as their name implies, are data centric. Responsibility-Driven Design proposes that, instead of thinking about data and algorithms which process data, one should think about objects with responsibilities.

Responsibilities are made up of two basic items: what it knows and what it can do. Basically, objects are bundles of data and operations on that date (Wirfs-Brock & Wilkerson, 1989).

Responsibilities are identified by highlighting the nouns and the verbs in the requirements, such as user stories. Verbs are candidates for actions a class can perform, and nouns are candidates for information that the class should maintain (Wirfs-Brock & Wilkerson, 1989).

In Responsibility-Driven Design, objects have a very specific part of the application. Each object is responsible for doing one portion of the work. Objects do only one job, and they must do that one job well. Objects then communicate with each other to fulfill the larger goals of the application.

Figure 1. Example sequence diagram

3.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) defines a standard set of diagrams used in designing software (Larman, Kruchten, & Bittner, 2001). UML sequence diagrams visually describe the actions of objects in a time sequence. An example of a sequence diagram is shown in Figure 1.

In the Unified Modeling Language, class diagrams show the relations and dependencies among classes. Class diagrams are used to show the overall architecture of a design. An example of a class diagram is shown in Figure 2.

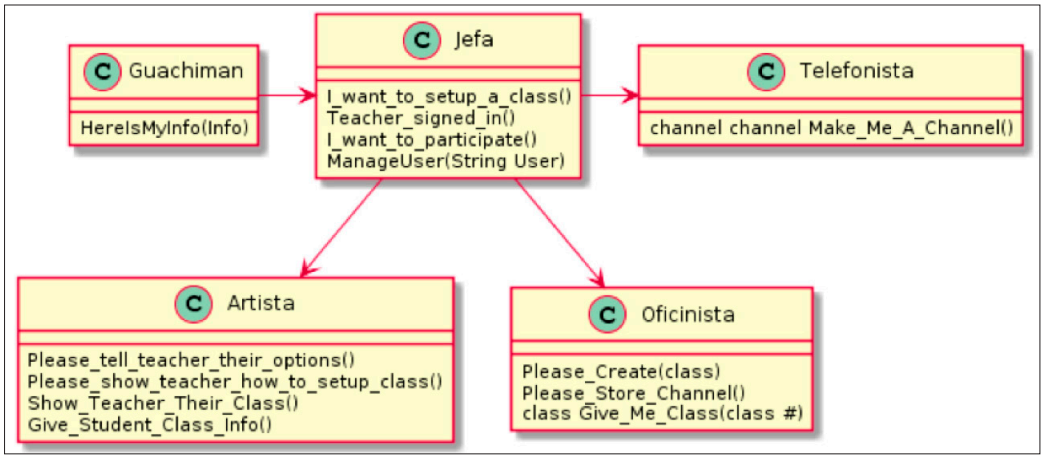

Figure 2. Example class diagram

4. Proposed analysis and design workflow

In this section, a simplified form of Responsibility-Driven Design is shown as a way for engineers to relate to software modules. For example, as a software engineer, you should try to describe how your software will work from the perspective of people doing the work, with the caveat that each person may do only one thing and must do it well. Put another way: write a play. A main character must be chosen, and conflict is needed (Dorf, 2018). In the world of software engineering, the main character is likely the human the software is being written for, and the conflict is what the main character is missing when the software does not exist. Developers must decide what roles are needed to resolve this conflict and the characters are given a significant problem to solve immediately (Dorf, 2018). Human personalities are given to the software modules. At this point, a beginning point is determined, and the engineering analysis comes from the story via actions and speech. Responsibility-Driven Design coupled with plays produces an analysis that is easily communicated to all stakeholders.

When the play is finished, it is then converted to a UML sequence diagram. Strict UML rules should not be enforced as the goal is to arrive at a candidate architecture for the software system. When the sequence diagram is complete, a UML class diagram can be made. Once again, strict UML rules should not be enforced. At this point, a candidate design which includes identified classes, associations, and method names is ready. An example of a play as well as related sequence and class diagrams are included in the appendices.

5. Results

To determine if this approach has merit, software design projects that were assigned to graduate students at the University of St. Thomas were analyzed. The Software Engineering beginning class has been consistently organized for the past three years. Students were required to form teams of two or three persons and design a major software project. Students were allowed to select the theme of their own projects, but in general students were guided towards projects where the user interface was a prominent part of the application. Some example projects include a WhatsApp-like application which translates texts to the native language of the receiver, classroom management applications, games, and medical-patient management systems.

In the first year studied (2015), the pre-play, students were asked to generate two-column use cases from user stories and then derive a design. Students generally had no trouble with the initial use case, which showed user and system interactions. These described "the user does this," "the system does that" type interaction. However, at this time many students were unable to identify the classes required to build a system. De-constructing the system into smaller objects was for many a frustrating task.

When assessing team progress, it was apparent that generally only one student in the group understood how to undertake this task adequately. This problem was reflected through summative assessment where nearly 50 percent of the students were unable to correctly create multiple two-column use cases, and then perform an analysis to derive required UML diagrams. Also, nearly 25 percent of the students who had created correctly two-column use cases and adequately identified classes were unable to properly suggest functions or methods within those classes. Informally, students also indicated frustration with this approach.

In mid-2016 performance, style plays where introduced as a method of analysis, and it became evident that the level of participation in the group activity rose dramatically. Performance style plays solved a significant problem facing the students the partitioning of the system object.

Students had less trouble identifying classes. Resolution of this difficulty was helped through the suggestion of characters with specialized skills for the play. Students now could envision a collection of specialists performing the tasks required for the system to operate. During the pre-play, it was difficult to envision how to divide up the work of the system. The post-play, which provided suggested characters by assigning tasks, became very practical and was no longer perceived as impossible.

Students also no longer had trouble identifying the methods required of each class, as methods were built upon the dialog between the characters in the play.

All members of the team took part in the creation of the play, and the post-play summative assessment rose to nearly 80 percent success. Thirty-nine final exams from two sections of pre-play classes and 119 final-exams from four sections of post-play classes were reviewed. Though the numbers of reviewed pre-play and post-play exams differed, the success percentages were consistent among classes. Additionally, post-play students who were not wholly successful were also not completely lost. In general, their issues were not severe. For example, "methods" might show up in the wrong class or "methods" might be missing. With a little bit more practice, these students could master this topic.

Table 3. *Successful projects*

Period	Value
Pre-play success	50%
Post-play success	80%

6. Conclusion

In this paper, a new approach to the analysis and design workflows is presented with the goal of avoiding a complicated software. Terminology and characteristics of a complicated software are provided. How to creatively perform the software engineering analysis and design workflows by writing plays inspired by "Responsibility-Driven Design" is shown. Information on creating UML sequence and UML class diagrams is given. Moreover, summative assessment is used as a measure of overall success and failure of the approach. Further research is warranted to analyze the most specific issues students had pre-play and how the performance style play could solve those issues. In addition, formal evaluation of large programming projects should also be done to verify good design quality, and good programming practice is necessary.

References

Army, U. (2015). Fm 6-0 commander and staff organization and operations. Washington.

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993, November). Software complexity and maintenance costs. *Commun. ACM, 36*(11), 81–94. DOI: 10.1145/163359.163375

Behe, M. J. (2009). Irreducible complexity: Obstacle to darwinian evolution. *Philosophy of biology: An Anthology*, 32, 427.

Dorf, J. (2018). *Playwriting 101 how to write a play*. Retrieved from http://www.playwriting101.com

Dorin, M. (2018). Coding for inspections and reviews.

Foote, B., & Yoder, J. (1997). Big ball of mud. Pattern languages of program design, 4, 654-692. Hall, R. (2012). Writing your first play. Focal Press.

Larman, C., Kruchten, P., & Bittner, K. (2001). How to fail with the rational unified process: Seven steps to pain and suffering. Valtech Technologies & Rational Software.

Rosenbloom, A. (2018). A simple MVC framework for web development courses. In *Proceedings of the 23rd western canadian conference on computing education* (pp. 13:1-13:3). New York, NY, USA: ACM. DOI:10.1145/3209635.3209637

Smith, R. (2010). The long history of gaming in military training. *Simulation & Gaming, 41*(1),6-19.

Victor, W. (2009). *Creative writing now, how to write a play*. Retrieved from https://www. creative-writing-now.com/how-to-write-a-play.html (Accessed: 2018-08-11)

Wirfs-Brock, R., & Wilkerson, B. (1989). Object-oriented design: A responsibility-driven approach. In *ACM sigplan notices* (Vol. 24, pp. 71-75).