System Dynamics Models and the Object-Oriented Paradigm

Warren W. Tignor PhD
Kimmich Software Systems, Inc.
7235 Dockside Lane
Columbia, Maryland 21045 USA
(410) 381-6009/(410) 381-5865 (fax)
wtignor@kssi.com

Abstract

The System Dynamics (SD) community recognizes that our craft, while powerful, is neither well-understood nor growing in popular acceptance. As we approach the gateway to the next millennium, we need to take the time to consider alternatives to our present approach to spreading the System Dynamics message. Generally, System Dynamicists will agree that their modeling work could benefit from more universal acceptance and understanding among its customers. If customers better understood the products developed for them, models would be prevalent and applied more diligently after delivery as opposed to, allegedly, often being set on the proverbial shelf to gather dust.

Introduction

To address the increasingly systemic problems faced by customers, there is a need to address numerous interrelated issues, e.g., the Greenhouse Effect, global population growth, and international finance. Yet, many System Dynamicists acknowledge that our field has not achieved wide recognition enabling discussion with customers of systemic topics such as these. Rather, customers continue to address very complex topics and make decisions using linear thinking and intuition as the norm.

System Dynamics and Software Engineering

It appears that our colleagues in the Software Engineering field, with regard to their craft, face many of the same issues as System Dynamicists. Their issues may actually be greater than ours with regard to customer acceptance and "buyin" of delivered products, at least measured in contract dollar value. Granted that feedback-based dynamic models may not be software engineering's forté; however, both System Dynamicists and Software Engineers in general develop "software-based" solutions to problems, i.e. models of things. Both professions seem to suffer the same dilemma of a lack of general customer and professional community acceptance.

In fact, to its credit, the Software Engineering field has documented evidence of the failure rate of its projects. Gibbs (1994) says that despite 50 years of progress, the software industry remains years or possibly decades short of the mature engineering discipline needed to meet the demands of an information-age society.

The Standish Group (Rational University, 1998) research shows that approximately 30% of software industry projects are canceled before completion. An estimated 50%

of the projects started cost nearly 190% of their original estimates. In 1995, the Standish Group estimated that American companies and government agencies spent $81 billion for canceled software projects. These same organizations were estimated to pay an additional $59 billion for software projects that were completed but exceeded their original budgets. Similarly, Arthur Anderson (Rational University, 1998) reported more than $300 billion was spent on commercial software activities in the U.S. and that only 8% resulted in software that was delivered and worked!

To remedy this problem, the software engineering community is adopting the Object-Oriented (OO) paradigm. Since the object-oriented paradigm captures system and software engineering work product in customer "language" for software engineering, it may be applicable to System Dynamics models as a means of gathering and communicating the requirements and the design of models. Assuming that "communication" is at least part of the reason that the SD customer doesn't understand System Dynamics models, perhaps a more universal customer-focused "language" would improve the communication?

Leveraging the object-oriented paradigm to System Dynamics models could lead to benefits such as more universally understood models, although the models themselves would remain embedded in the causal loop, and stock and flow language essential to the principles of System Dynamics. The object-oriented paradigm and Universal Modeling Language (UML) offer an opportunity as a meta-language for the definition and design of System Dynamic models.

The OO paradigm captures system and software engineering work product in frameworks of packages, classes, objects, and methods. The language of the customer is captured by Use Cases as a statement of need and concept of operation. Leveraging the OO paradigm to System Dynamics models will lead to benefits such as better understood: models, software design, and reusable software model libraries

Hypothesis

The remainder of this paper will address the hypothesis that what the Software Engineering industry is doing through the use of the OO paradigm and UML to start to remedy its problems will apply to System Dynamics modeling. OO and UML will provide a basis of communication with the customer and among System Dynamicists as a meta-language for models.

Background

Interestingly, our colleagues in the discrete simulation and modeling world have already recognized the opportunity to use the OO paradigm and UML, and they are acting on it. Braude (1998) applied recent advances in object-oriented research to propose a "class-level" framework for discrete simulations. Schöckle (1994) took an "object-oriented" approach in his work with modeling systems.

Braude (1998) says that there has been relatively little sharing of code or design for discrete simulation systems. Sharing, if any, has typically occurred at the tool level by means of commercially available graphics-based environments for building simulations (Braude, 1998).

Based on the maturity of the OO paradigm and the adoption of UML, he believes that the time has arrived to attempt to design a standard framework for discrete simulations, Braude (1998). He cites the definition of an application framework as a reusable, "semicomplete" application that can be specialized to produce custom applications, (Fayad & Schmidt, 1997).

Schöckle (1994) says that the OO paradigm offers several possibilities not available in the traditional procedural programming approach, which help to deal with complex systems:

- OO building blocks are "objects" which encapsulate functions and data.
- Procedural building blocks are "procedures" which only abstract their functions.

Additionally, the OO paradigm provides concepts for managing complexity not available in procedural environments: classes, inheritance, polymorphism, and communication of messages (Schöckle 1994).

Keys to Object-Oriented Technology

Taylor (1990) identifies three keys to understanding the object-oriented paradigm, i.e., objects, messages, and classes. According to Taylor (1990), the concept of software objects came from the need to model real-world objects in computer simulations. For example, SIMULA, created by O. J. Dahl and Kristen Nygaard of Norway, builds accurate working models of complex physical systems containing thousands of objects, Taylor (1990).

An object is software that contains a collection of related procedures and data, (Taylor, 1990). In the object-oriented approach, procedures go by a special name; they are called methods. In keeping with traditional programming terminology, the data elements are referred to as variables because their values can change over time, (Taylor, 1990).

Real-world objects can have an unlimited number of effects on each other, e.g., create, destroy, lift, attach, buy, sell. The way objects interact is by sending messages to each other. "A message is simply the name of an object followed by the name of a method the object knows how to execute, (Taylor, 1990, p. 19)". Taylor (1990) adds that if a method requires any additional information in order to execute, the message includes that information as a collection of data elements called parameters.

Since most software systems or simulations will have a plethora of objects, methods, and variable as opposed to a single object with its methods and variables, the concept of class was created. "A class is a template that defines the methods and variables to be included in a particular type of object. The descriptions of the methods and variables that support them are included only once, in the definition of the class. The objects that belong to a class, called instances of the class, contain only their particular values for the variables, (Taylor, 1990, p. 20)".

Software as Simulation

According to Taylor (1990) software makes a computer perform a task. The typical progression of software development projects reflects this view, e.g., specification of the problem to be solved, design of a system that solves the problem. The result usually is a tailored system that performs the original task but is ill suited to handling any others, even when dealing with the same real-world objects. For example, a billing system is not able to handle mailings for marketing or reminders for the sales team (Taylor, 1990). Taylor (1990) says that if you have a good OO model of your customer's interactions, the model will be equally useful for billings, mailings and reminders.

Keys to System Dynamics Technology

System Dynamics is a rigorous method for qualitative description, exploration and analysis of complex systems in terms of their processes, information, organizational boundaries and strategies, which facilitate quantitative simulation modeling and analysis for the design of the system structure and control, Wolstenholme (1990).

In the field of System Dynamics, a system is defined as a collection of elements which continually interact over time to form a unified whole, (Halbower, 1994). The pattern of interaction among the system elements is the system's structure. The term dynamics refers to the system's change over time.

From a computer simulation perspective, STELLA is one of many software programs that provides a graphical interface for observing the quantitative interaction of variables within a system. STELLA models, like most SD models, are made up of four building blocks; these building blocks are defined as follows (Halbower, 1994, p. 12):

STOCK - "A stock is a generic symbol for anything that accumulates or drains." Water accumulates in a sink; the amount of water in the sink is the stock of water.

FLOW - "A flow is a rate of change of a stock". A flow is the water coming into the sink through the faucet and the water leaving the sink through the drain.

CONVERTER - "A converter is used to take input data and manipulate or convert that input into some output signal". In the sink example, turning the valve which controls the water flow in the sink, the converter takes as input the action of turning the valve and converts that signal into an output reflecting the flow of water.

CONNECTOR - "A connector is an arrow which allows information to pass between converters and converters, stocks and converters, stocks and flows, and converters and flows".

Senge (1990) tells us that a language for talking about the complex structures that evolve from constructing models from the elements enumerated above is the language of complexity. He prefers to use system archetypes to “objectify” the fundamental systemic forces at play.

Simulation as Software

Procedural, non-object-oriented, software is written to solve a specific problem. Whereas, object-oriented software models a system, (Taylor, 1990). Taylor (1990) advocates that there is a different mindset underlying object-oriented technology. He concedes that the technology has spread far beyond its origins as a simulation language (SIMULA). However, programming with objects still retains the spirit of real-world simulation, (Taylor, 1990). To Taylor (1990), the design of an object-oriented system begins with the aspects of the real world that need to be modeled in order to perform that task. This is in contrast to the procedural approach that starts with the task to be performed.

Abstraction and Computer Language

Eckel (1998) discusses programming languages as abstractions, to consider apart from application to or association with a particular instance. He asserts that the complexity of the problem one can solve is directly related to the kind and quality of abstraction (Eckel, 1998). To Eckel (1998), assembly language is an abstraction of the underlying machine. Languages such as FORTRAN, BASIC and C are abstractions of assembly language according to Eckel (1998). These languages require one to think in terms of the structure of the computer rather than the structure of the problem to be solved. The “programmer” establishes the association between the machine model (the “solution space”) and the model of the problem that is actually being solved (the “problem space”), Eckel (1998). According to Eckel (1998), this results in programs that are difficult to write and expensive to maintain, and as a “side effect” created the “programming methods” industry due to:

1. The effort required to map the “solution space” and the “problem space”
2. The fact that the mapping is extrinsic to the programming language.

The alternative to modeling the machine, i.e. the computer, is to model the problem being solved; this is the point of the OO paradigm.

Eckel (1998) states that the OO paradigm goes a step further by providing tools for the programmer to represent elements in the problem space. This representation doesn’t constrain the programmer to any particular type of problem. The elements in the problem space and their representations in the solution space are called “objects”. The notion is that the program is allowed to adapt itself to the “lingo” of the problem (customer’s language) by adding new types of objects. This means that when one reads the artifacts describing the solution, one reads words that also express the problem in the customer’s language.

UML and Problem Description

When Grady Booch, Ivar Jacobson and James Rumbaugh began crafting the Unified Modeling Language, they aimed to produce a standard means of expressing design that would reflect the best practices of industry, and also demystify the process of software system modeling (Fowler, 1997). They believed that the availability of a standard modeling language would encourage developers to model their software systems before building them (Fowler, 1997).

Fowler says that one of the biggest challenges in software development is building the "right" system that meets the customer's needs at a reasonable price. To Fowler (1997), achieving good communication with the customer, and an understanding of the customer's world is key to developing good software. To this end, Fowler recommends the Use Case, a snapshot of one aspect of a system.

Use Case

A good collection of Use Cases is key to understanding what the customer wants, Fowler (1997). They also present a good vehicle for project planning as they help control iterative development by identifying the elements of the system as a whole that can be scheduled for completion over time.

In essence, a Use Case is a typical interaction between a customer and a computer system, Fowler (1997):

- Captures some user-visible function
- May be small or large
- Achieves a discrete customer goal.

Use Cases are captured by talking to the customer and discussing the various things that they might want to do with the system or understand about the system. Each discrete thing the customer wants to do or understand receives a name and approximately a paragraph of short textual description, Fowler (1997).

Jacobson introduced a diagram for visualizing Use Cases; the diagram is now part of UML (Fowler, 1997). The Use Case diagram shows the system Actors, where an Actor is a role that a customer plays with respect to the system, Fowler (1997). Actors perform Use Cases. Actors do not need to be humans; they can also be an external system that needs some information from the current system. Actors are not part of the system, but represent anyone or anything that must interact with the system being developed, (Quatrani, 1998).

A Use Case diagram is a graphical view of Actors, Use Case, and their relationships for a system, Quatrani (1998). He says that typically each system will have a main Use Case diagram which will picture the system boundary (Actors) and the major functionality provided by the system (Use Cases), Quatrani (1998).

Quatrani (1998), like Fowler (1997), says that the most important role of a Use Case is one of communication by illustrating the system's:

- Intended functions (Use Case)

- Surroundings/Interfaces (Actor)
- Relationships between Intended functions and Surroundings/Interfaces (Use Case Diagram).

The Use Case provides a vehicle for the customer and the developer to discuss the system's functionality and behavior, Quatrani (1998, p. 21).

In sum, the collection of Use Cases for a system constitutes all the defined ways the system may be used (Quatrani, 1998). Quatrani provides a formal definition of a Use Case as "…a sequence of transactions performed by a system that yields a measurable result of value for a particular Actor", (1998, p. 25).

He says that a Use Case consists of:

- A brief description that states its purpose in a few sentences and provides a high-level definition of functionality
- A documented flow of events needed to accomplish the required behavior.

A suggested template to complete a Use Case definition over the course of system definition follows (Quatrani, 1998):

- N.1 Brief Description of <name> Use Case
- N.2 Flow of Events for <name> Use Case
- N.2.1 Preconditions
- N.2.2 Main Flow
- N.2.3 Subflows (if applicable)
- N.2.4 Alternate Flows

Class Diagram

As Use Cases help communicate about surface things, it is Class Diagrams that look at the deeper things, Fowler (1997). Fowler (1997) advocates using Class Diagrams in a conceptual manner, initially treating each class as a concept in a customer's mind and in the customer's language. Later as the design is "fleshed-out" the conceptual classes transition to physical representations of the solution.

Activity Diagram

Fowler (1997) points out that Activity Diagrams are very useful in cases where workflow processes are an important part of the customer's world. Activity Diagrams support parallel processes and de-emphasize links to classes.

Package Diagram

To build a system roadmap, Fowler (1997) uses Package Diagrams at the high levels of the design to scope out a Class Diagram. When drawing a Class Diagram for a roadmap, it takes a specification perspective. Fowler (1997, p. 10) says that it is very important to "hide implementations" with this kind of work.

Pattern

Lastly, "patterns" describe the key ideas in the system, Fowler (1997). Patterns help explain why a design is the way it is. The design pattern represents the fundamental algorithm being implemented by the software, an algorithm that is repeated in many other designs. Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995) characterize design pattern as a description of communicating objects and classes that are customized to solve a general design problem in a particular context. The design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.

Summary

To summarize, System Dynamics modeling and software engineering have common problems of acceptance by customers. The software engineering profession developed a paradigm, OO, and a modeling language, UML, to help it communicate with its customers. This paper attempts to draw parallels between System Dynamics modeling and software engineering with the hypothesis that leveraging the OO paradigm and UML will improve communication with System Dynamics' customers and lead to better acceptance of products. The table below is a brief summary of the OO and SD "toolset" and their analogous counterparts:

OO/SD	Description of the Problem	Causal Diagram	Stock and Flow Diagram	Archetype	Model Code
Use Case	X				
Use Case Diagram	X				
Class Diagram		X	X		
Package Diagram			X		
Activity Diagram			X		
Pattern				X	
Code					X

Conclusion

Based on the referenced material, there is a strong association between System Dynamics modeling and software engineering. There are also similar problems with customer acceptance of the work products generated by both System Dynamicists and

Software Engineers. The OO paradigm and UML, created for Software Engineering, offers us a tool to improve communication with our customers and among ourselves. Our colleagues in the discrete simulation field have begun to apply OO and UML to their problem domain. It appears appropriate to consider the merits of OO and UML for the System Dynamics field.

Bibliography

Braude, E. (1998). Towards a standard class framework for discrete event simulation. Proceedings of 31st Annual Simulation Symposium, pp. 4-8.

Eckel, B. (1998). Thinking in java. Upper Saddle River: Prentice Hall.

Fayad, M., & Schmidt, D. (1997, Oct.). Object-oriented application frameworks. Communications of the ACM 40.

Fowler, M. (1997). UML Distilled: Applying the standard object modeling language. Reading: Addison-Wesley.

Gibbs, W. (1994, Sept.). Software's chronic crisis. Scientific American, pp. 86-95.

Halbower, M. (1994, Dec.). The first three hours: An introduction to system dynamics through computer modeling. (Available at http://sysdyn.mit.edu/road-maps/rm-toc.html)

Object-oriented analysis and design with C++ student manual (part1 of 2) part# 800-011426-000, version 3.6: (1998). Rational Software Corporation. (Available from Rational University, 2800 San Tomas Expressway, Santa Clara, CA 95051)

Quatrani, T. (1998). Visual modeling with rational rose and uml. Reading: Addison-Wesley.

Senge, P. (1990). The fifth discipline. New York: Doubleday.

Schöckle, M. (1994). An object oriented environment for modeling and simulation of large continuous systems. (Available at http://www.nmr.embl-heidelberg.de/eduStep/…erences/OOCNS94/Proceedings/Schoeckle.html)

Taylor, D. (1990). Object-Oriented Technology: A manager's guide. Reading: Addison-Wesley.

Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). Design Patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley.

Wolstenholme, E. (1990). System enquiry: A system dynamic approach. Chicester: Wiley & Sons.