Cybernetic Components:
A Theoretical Basis for Component Software Systems

Charles Herring and Simon Kaplan

Department of Computer Science and Electrical Engineering
The University of Queensland, St Lucia QLD 4072

Abstract

We take the position that Cybernetics offers a theoretical basis for the development of components, component frameworks, and component framework frameworks. Although not recognized as such, Cybernetic principles are currently used in successful software. A goal in proposing Cybernetics for component design is to permit the development of organic software systems to meet increasingly complex application requirements.

Introduction

Apparently the component-oriented approach to software construction, deployment, and organization when coupled with economic viability represents the next stage in the evolution of software practice. Szyperski [1] observes that components are an improvement over objects for building large systems, but objects are probably the best way to realize components due to the *modeling advantages* of object technology. He goes on to say:

> *On the flip side, modeling of component-based systems is still a largely unresolved problem.*

Thus, while components are an improvement over objects and solve significant technical and social problems related to large systems, they do not address problems central to software engineering. However, components move the state of the practice of software closer to becoming an engineering discipline. This is particularly important in that we are attempting to build ever more complex systems.

Software Engineering is currently an aggregation of disjoint tools, techniques, and methodologies that more or less works depending on the nature of the problem, the support infrastructure, and the skill of the practitioner. Engineering, as the application of scientific knowledge, must rest on some scientific basis. Software Engineering has struggled to find such a footing. There is no "Theory of Software" backing up an engineering approach to the design, construction and operation of large scale, component systems. The position taken up here is that Cybernetics can provide such a theoretical underpinning and permit an engineering approach to the design (modeling) of component systems. Further, such a basis is required to meet the challenges posed by increasingly complex system requirements.

Forward to the Past

Von Neumann's 1945 report outlining the construction of EDVAC (the first "von Neumann machine") was described in biological terms [2]. He designed a machine architecture based on the idealized neurons of McCulloch and Pitts and compared the desired function of the machine to that of the human nervous system (asynchronous and parallel). He continued with this biological analogy until his early death in 1957.

Von Neumann worked toward the develop of a general theory of information processing that would be the basis for understanding biological processing and serve as the foundation for computing. His program to accomplish this proceeded as follows. First he constructed a logical theory of automata based on McCulloch, Pitts and Turing. Next he investigated how this logic theory could be developed into a probabilistic theory of automata based on Shannon (communications theory) and Wiener (cybernetics). The need for a probabilistic theory was motivated by the desire to model the complexity of biological and mechanical systems that demonstrate *reliable processing based on unreliable components operating in a dynamic environment*.

So after a considerable hiatus (Artificial Intelligence and "Complexity Theory" not withstanding) we resume the direct line of investigation that began with von Neumann into a Cybernetic basis for the organization of computation. We are fortunate in having the last fifty-odd years of experience, however, our task is no less difficult as our ambitions for software systems is much greater.

Cybernetics

Norbert Wiener coined the term Cybernetics around 1948 from the Greek word meaning "steersman" [3]. He defined Cybernetics as the science of communication and control in mechanisms, organisms and society. Cybernetics is a general theory of control that can be applied to any organized system. Several important findings about control systems were made early. The "First Law of Cybernetics" says that any system that seeks to control another system (including self) must have a model of that system. The role of information in control systems and the concept of *feed-back* were major discoveries.

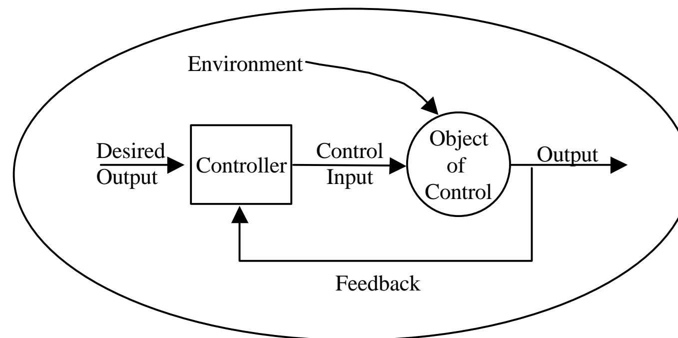

Figure 1 Closed-loop feedback control system

A prototypical control system is shown in Figure 1. Note the system is embedded in an environment that effects (disturbs) the system. Control systems fall into four major categories: stabilizing, following (tracking), programmed and optimizing. Also, Cybernetics deals with adaptive and self-organizing systems wherein the system changes itself in order to achieve better performance (survive) in a changing environment. A handy on-line reference to Cybernetics is [4].

The Canonical Design Problem

Shaw [5] analyzes the "canonical" object-oriented design problem – the automobile cruise control. Her fundamental insight is that *it is a control problem* and there is a body of engineering knowledge associated with such systems. She argues that the object-oriented paradigm may not be the most suitable approach in this case. She goes on to develop a software architecture based on classical feedback control theory as opposed to one based on identification of objects and their relationships. A diagram of the cruise control system is shown in Figure 2.

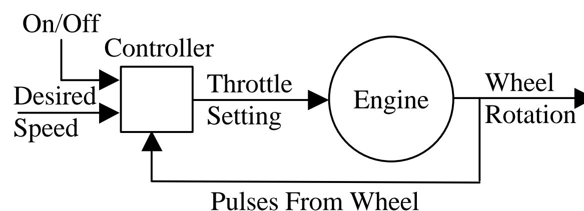

Figure 2 Control Architecture for Cruise Control

Shaw offers the following "rule of thumb" for use of her "control paradigm" architecture:

> *When the execution of a software system is affected by external disturbances, forces, or events that are not directly visible to, or controllable by, the software this is indication that a control paradigm should be considered for the software architecture.*

Shaw says the control paradigm separates the *operation* (the Engine in this case) of the main process from the compensation for external disturbances, the *control*. This separation of concern yields appropriate abstractions that leads to design issues that might otherwise to missed. In particular, performance and correctness constraints are identified early in the design process.

One of the main reasons problems become difficult is they are formulated in a manner that frustrates solution. This is what Shaw has seen. The object paradigm is not the correct way to analyze this problem. Software architecture is concerned with the fundamental components that make up a system, how these components relate to each other and how the system will evolve over time. Object analysis does not lead to a suitable architecture for this particular problem. However, it may be a useful aid in subsequent steps of implementing the software architecture.

In analyzing the cruise control problem Shaw recognized that control theory was the most appropriate context for the problem. We argue that her rule of thumb is the general case for complex software systems. We maintain that software systems should be designed in anticipation of being *affected by external disturbances*. Cybernetics offers a principled approach to doing just that, and if its principles are in fact general there is no choice – they cannot be avoided.

The Canonical Framework Pattern

The Model-View-Controller (MVC) triad is perhaps the most successful and familiar design pattern. It is used as a basis for many types of object-oriented applications and frameworks. MVC has been extended, enhanced and augmented in various ways. Its success, we claim, is because it is a *closed-loop feedback control system*, although never described in those terms before.

We now provide an analysis of MVC as a control system. Figure 3 shows the major components and their relationships. In the diagram, the conventional names (model, view, and controller) are in bold. Their Cybernetic counterparts are in parenthesis. The Model as the Object of Control is obvious. The Controller is more precisely identified as an Actuator. In our refactoring of MVC, the Controller is a combination of the User and the View (shown in a square).

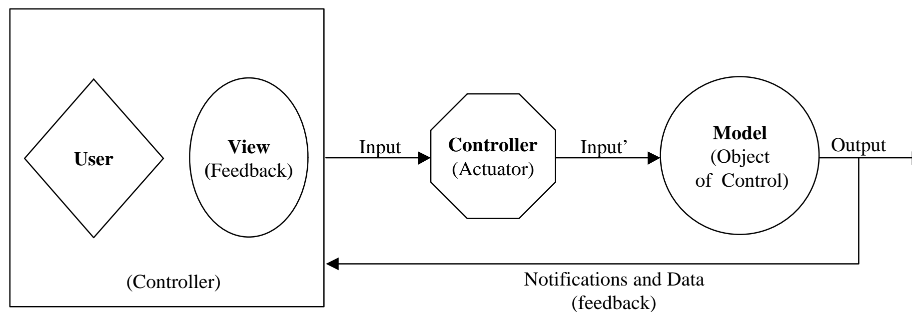

Figure 3 MVC as Control System

The Actuator takes input from the Controller and transforms it into signals the Model can accept. In the case of a user interface implementation of MVC, the Input (from Controller to Actuator) is in the form of events (generated from human-computer input devices) and is translated into Model-specific commands (input' in the diagram). Signals generated by the Controller (transformed and transmitted via the Actuator) cause changes in the Model and results in signals (feedback) being sent back to the Controller. Two classes of information are carried on the feedback channel: notifications and data. Notifications tell the Controller that the Model has changed state and needs to be updated with new data. Data is retrieved and used as feedback to further drive the state of the Model.

The User is not considered a component of the MVC system - otherwise it would be called UMVC. In the Cybernetic analysis, the User is part of the Controller and constitutes the decision-making function that dynamically directs state changes in the Model. The View is a transducer that transforms the Model's

information signals into a usable representation. Note the User is not necessarily human. In some implementations another Model may be the "User."

Cybernetics is inherently comprehensive in that it includes the Environment. This is implied in the MVC diagram and shown explicitly in the other figures above. The environment may act on any part of the system. In MVC the Model is affected by the presence of other View-Controller pairs in the environment of an application. There will also be system or application dependent forces acting on the Model as well. It is the Controller's task to cope with these disturbances.

The design of MVC is motivated by a number of factors. First, in the context of a graphical user interface framework, there is the need to separate the machine dependence of graphical presentation specifics from the underlying application logic. This permits for ease of porting the framework. Also, the separation of interface from application permits the development of different (and multiple concurrent) presentations that use the same model. The forces driving these design decisions are the same ones that result in the separation of control from the object of control in Cybernetics. We believe the designers of MVC were subconsciously applying the general principles that Cybernetics has identified.

Toward Cybernetic Components

We have taken some first steps in applying Cybernetics to the design of component systems. To get an idea of what a "Cybernetic Theory of Components" might look like, we examined two Cybernetics-based approaches: Stafford Beer's Viable System Model (VSM) and James Miller's Living Systems Theory (LST). Please note, we do not claim these models are directly applicable to component software development. They are Cybernetics-based methodologies built for specific domains. We do maintain the underlying principles and insights, when cast into a software specific design methodology, will be beneficial.

Stafford Beer's VSM was developed to aid in the analysis and understanding of complex organizations. His book "Diagnosing the System for Organization" provides the most concise exposition of the model and is an excellent example of a Cybernetics-based methodology [6]. A Viable System is able to maintain a separate existence, a separate identity. This is a characteristic of biological entities. It is also a characteristic of successful organizations. An organization is viable if it can survive in its environment with some degree of autonomy. VSM is a tool to help understand organizations and diagnose to what degree they are or can become viable. We seek to design component "organizations" which demonstrate characteristics of viable systems. Our preliminary thoughts on VSM applied to components are recorded at [7] and we do not go into more detail in this paper.

In the remainder of this section we dwell on LST. We give a brief overview of the most import concepts and provide examples of how the elements of LST may be applied to component system design and understanding. Miller's life work has been the development of a general theory of living systems. His magnum opus, *Living Systems* [8], first published in 1978, presents a general model of biological systems and applies it to the range of living systems from the cell to the supranational entity. LST is a system-of-systems model of biological organisms and organizations. It considers them to be conceptually divided into eight levels:

1. Cell
2. Organ
3. Organism
4. Group
5. Organization
6. Community
7. Society
8. Supranational

At each level, systems are composed of 20 critical subsystems. The subsystems are grouped in terms of how they process matter-energy or information. Eight sub-systems process matter-energy, ten process information, and two process both. They are as follows:

Subsystems that process both matter-energy and information

1. Reproducer (Re): responsible for replication and evolution.
2. Boundary (Bo): contains, protects, and permits entry and exit of matter-energy and information.

Subsystems that process matter-energy

3. Ingestor (IN): brings matter-energy across boundary.
4. Distributor (DI): carries inputs from outside or outputs from within to subsystems.
5. Converter (CO): transforms certain inputs into internal formats.
6. Producer (PR): synthesizes material for growth, repair or replacement.
7. Matter-energy storage (MS): stores and retrieves matter-energy
8. Extruder (EX): transmits products or wastes out of the system.
9. Motor (MO): moves the system or parts in relation to environment.
10. Supporter (SU): maintains spatial relationships of subsystems.

Subsystems that process information

11. Input Transducer (it): sensory subsystem that brings information markers in and transforms them into internal formats suitable for transmission.
12. Internal Transducer (in): receives internal subsystem information markers and transforms them into internal formats suitable for transmission.
13. Channel and net (cn): provides for transmission of markers.
14. Timer (ti): transmits to decider information about environment or internal subsystem states.
15. Decoder (dc): transforms code of marker into private format.
16. Associator (as): first stage of learning and forms associations.
17. Memory (me): second stage of learning, stores and retrieves information.
18. Decider (de): receives inputs from and transmits control information to all subsystems.
19. Encoder (en): transform privately coded information into public coded format.
20. Output transducer (ot): transforms and outputs information markers.

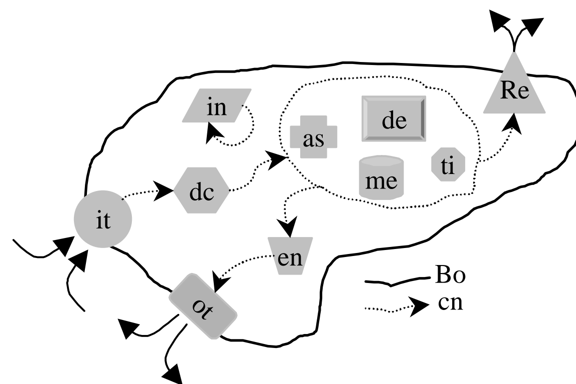

Figure 4 Prototypical unit (information processing subsystems only)

A simplified LST unit is shown in Figure 4. Here only the subsystems that process information are shown. The information bearing relationships, Channel and Net (cn), are also shown. The Internal Transducer (in) is shown with a channel to itself. This is to indicate that any or all of the subsystems may call up its functions. Also, the Executive subsystem (as, de, me and ti) are shown as a group. All of these subsystems interact with each other (and the other subsystems) in the regulatory (feedback) and decision making functions.

In applying LST to component software there are at least two possible ways to proceed. The first is to develop a component system based on the theory and the second is to examine an existing system to see how it compares. While the former may eventually have to be done to verify the approach, the latter is a valid fist step. This is akin to the "patterns" approach. Going down this path, we will look at some existing systems and see if they exhibit the "LST pattern."

Design Patterns

We investigated the "Gang of Four" design patterns catalog [9] in light of the LST critical subsystems. The GOF design patterns are divided into three categories: Creational, Structural, and Behavioral. Four of the patterns apply to classes (Factory Method, Adapter, Interpreter, and Template Method) and the others apply to objects.

LST	Creational	Structural	Behavioral
Reproducer (Re)	Abstract Factory Builder, Factory Method Prototype		
Boundary (Bo)		Composite, Façade	
Input Transducer (it)		Command	
Internal Transducer (in)		Adapter, Proxy	
Channel and net (cn)		Chain of Responsibility Observer Mediator	
Timer (ti)			
Decoder (dc)		Interpreter	
Associator (as)		Bridge	
Memory (me)			Memento
Decider (de)			Strategy
Encoder (en)			State
Output transducer (ot)		Command	

Table 1 LST and GOF Design Patterns

A quick comparison of the structure and functions of the GOF patterns and the LST critical subsystems yielded Table 1. There seems no obvious mapping of the following design patterns: Singleton, Decorator, Flyweight, Iterator, Template Method, and Visitor. These patterns appear to be implementation artifacts rather than modeling constructs. The GOF design patterns are NOT a pattern language and do not claim to be. In that regard they are only loosely connected. Notice the absence of anything in the Timer category. It would be an interesting exercise to put the LST subsystems into pattern form. Would they constitute the LST pattern language?

Microsoft's Distributed Component Architecture

Microsoft's Distributed Component Architecture (MDCA) is an example of a large multi-layer component system [10]. The major pieces of MDCA are:

- COM (Component Object Model) which defines the basic component/object model.
- DCOM (Distributed COM) which allows components to be moved around a network.
- MTS (Microsoft Transaction Server) which provides a component runtime environment for the middle tier. It provides a container for server-side components.
- DTC (Distributed Transaction Coordinator) which coordinates distributed transactions.
- MSMQ (Microsoft Message Queuing) which provides asynchronous communications.
- MSCS (Microsoft Cluster Server) which allows multiple servers to work as one.

Of course all these run on the NT operating system and how best to arrange these into levels is not entirely clear. Many of these systems cut across levels or may be orthogonal. As Szyperski points out, the levels (or tiers) in a component system are not the “Three-tier” architecture of client, business logic, and back-end databases. Here, the tiers are component, component framework, and component system (framework for frameworks). Following LST, the levels of the system under consideration must be specified. The mapping between LST levels and MDCA may not be perfect, but let us try this:

- Cell — Internals of COM components
- Organ — COM Component (COM, OLE, ActiveX)
- Organism — Component container (e.g. OLE container, Word, IE4)
- Group — Office 2000 (collaborative component application suit)
- Organization — MTS (container of server-side components)
- Community — MSCS, DTC and MSMQ (cluster of machines)
- Society — Enterprise (?)
- Supranational — The Web (?)

In our analysis of Microsoft we focused on the Cell, Organ, and Organism levels [11]. To summarize our findings, in a system such as MCDA, or any complex software system, we think most, if not all, of the critical subsystems will be found. (Please refer back to the list of critical subsystems and Figure 4 now.) However, they may not be organized along the lines of a living system. In particular, the Associator and Decider functions may be trivial or dispersed, as is the case with the biological cell. Also the Timer and its function may be absent. But the basic subsystems will be present: Boundary, Input, Output, Encoder, Decoder, Internal Transducer, Reproducer, Memory and Channel. LST offers a “language of components” in which to talk about design and organization of these subsystems and the hierarchy of subsystems that must exist in complex systems.

Model-View-Controller

A possible overlay of the fundamental Model-View-Controller pattern of objects (or components) on top of the prototypical LST unit is shown in Figure 5. As remarked above, the Controller is more precisely called an Actuator. This view might be called the “Behavioral Psychology” model in that the LST unit is the object of control by an external controller. If the Controller and View are merely (sensory) Input and Output, then the control function must resides inside the unit and it might be called the “Self Actualizing” (Agent?) model. Much remains to be done here.

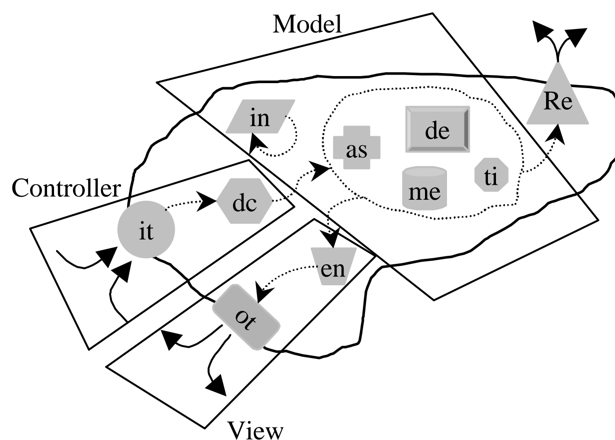

Figure 5 MVC meets LST

Summary

We take the position that design and organization of component software systems can benefit from the principles of Cybernetics. We have provided some background to this, possibly unfamiliar, science and described our preliminary results. We close with one concrete insight generated from our investigations and some general remarks.

The Timer was a late addition to the critical subsystems of Millers' LST appearing in the second edition of his book. It is interesting that he overlooked this subsystem as timing is so crucial to living systems, e.g. heart beat, daily and yearly cycles. Also, component and object systems tend not to have timers per se. Most components are simple procedural programs. Szyperski (Chapter 22, Section 1.3 "Learning from Circuit Design") discusses the possibility of synchronous clock-driven components versus the common invocation-driven design. Instead of using processes the computation can be organized based on more lightweight abstractions. He calls these *atomic actions* and they are triggered by events queued in the system and there is always only one action executing. We have proposed (designed and implemented) similar lightweight process facilities for this purpose in Java [12]. These features permit a form of "temporal programming" that permit modeling complex systems and their interactions in the "process-based discrete-event" simulation style. We argue this capability must be available at the language level. Achieving "organic" component behavior is closer to discrete-event simulation that might be expected. Without these language constructs temporal programming in this style is essentially prohibited.

From the earliest days of computing, there was the desire to endow hardware and software with the characteristics of the biological systems: robust, fault-tolerant, self-organizing, self-repairing, etc. As we seek to build ever more complex systems, the need to achieve these goals become even more important. Not so long ago a statement like "Programs = Algorithms + Data" seemed comforting. The familiar underlying basis of discrete mathematics, automata and information theory gave assurance that we knew what we were doing. Then came the need to build larger more challenging systems and the "illities" of Software Engineering emerged to guide us on the narrow path. While everything that has gone before is useful, it falls short of the inspiration provided by the biological systems. To that end we advance (a return to) Cybernetics in the hope that it, coupled with the new technology of components, may help realize the original dream.

Reference

1. Szyperski, C., *Component Software*. 1998, New York: Addison-Wesley.
2. Aspray, W., *The Origins of John von Neumann's Theory of Automata*, in *The Legacy of John von Neumann*, J. Glimm, J. Impagliazzo, and I. Singer, Editors. 1990, American Mathematical Society: Providence, Rhode Island. p. 289-309.
3. Wiener, N., *Cybernetics*. 1948, New York: Wiley and Sons.
4. Principia Mathematica Web, *http://pespmc1.vub.ac.be/Default.html.*
5. Shaw, M., *Beyond objects: A software design paradigm based on process control.* ACM Software Engineering Notes, 1995. **20**(1).
6. Beer, S., *Diagnosing the system: for organisations*. 1985, Great Britain: John Wiley and Sons Ltd.
7. Herring, C., *Viable System Model and Component Software.* http://www.dstc.edu.au/TU/staff/herring/manifolds.htm, 1998.
8. Miller, J.G., *Living Systems*. 1995, Niwot, Colorado: University Press of Colorado. 1102.
9. Gamma, E., *et al.*, *Design Patterns*. 1995, New York: Addison-Wesley.
10. Sessions, R., *COM and DCOM: Microsoft's Vision for Distributed Objects*. 1998, New York: John Wiley and Sons.
11. Herring, C., *Living Systems Theory and Component Systems.* http://www.dstc.edu.au/TU/staff/herring/manifolds.htm, 1998.
12. Herring, C., *Proposal to include process-based discrete-event simulation capabilities in Java.* http://www.dstc.edu.au/TU/staff/herring/java.html, 1997.