Regularities in Software Systems

By: Naftaly H. Minsky

April 1993

Computer Science Department[1]
Rutgers University
New Brunswick, NJ 08903
Net Address: minsky@cs.rutgers.edu

[1]Work supported in part by NSF grant No. CCR-8807803.

Abstract

Regularities, or the conformity to unifying principles, are essential to the comprehensibility, manageability and reliability of large software systems. Yet, as is argued in this paper, the inherent globality of regularities makes them very hard to establish in traditional methods, unless they are built into the very fabric of a programming language.

This paper explores an approach to regularities which greatly simplifies their implementation, making them more easily employable for taming of the complexities of large systems. This approach, which is based on the concept of law-governed architecture (LGA), provides system designers and builders with the means for establishing regularities simply by declaring them formally and explicitly as *the law of the system*. Once such a *law-governed regularity* is declared, it is enforced by the environment in which the system is developed. Although not all desirable regularities can be established this way, it is argued that the range of feasible "law-governed regularities", which can be easily defined and efficiently enforced, is sufficiently broad for this to become a useful software engineering technique.

1. Introduction

In his classic paper "No Silver Bullet" [2], Brooks cites *complexity* as a major reason for the great difficulties we have with large software systems, arguing that "software entities are more complex for their size than perhaps any other human construct", and that their "complexity is an *inherent* and *irreducible* property of software systems" [emphasis mine]. Brooks explains this bleak assessment as follows: "The physicist labors on, in a firm faith that there are *unifying principles* to be found ... no such faith comforts the software engineer."

Brooks is surely right in viewing conformity to unifying principles, i.e., *regularities*[2], as essential to simplicity in large systems; but he is overly pessimistic about the role that regularities can play in software systems. It is true that one is not likely to find many regularities of *repetition* in software, because plain repetitions can be easily abstracted out and "made into a subroutine", in Brooks' words. But there are other, more subtle kinds of regularities, that may "comfort the software engineer", provided that they can be easily and reliably established, or implemented.

What we mean by the term "regularity" in this paper is any structural or behavioral property that holds true for the entire system, or, at least, for a significant, well defined part of it. Such regularities are absolutely essential for the comprehensibility of large systems of any kind. It is, in particular, due to its regularities that nature is as comprehensible and as manageable as it is. Regularities are being used for software systems as well, rendering them simpler, more manageable and more secure. A case in point is *encapsulation* -- the principle that no object in a system can penetrate the interior of another object. This regularity is the basis for meaningful modularization, and is thus vital for large systems. Another example, is *layered-structure* [21] -- the grouping of the modules of a system into "layers", together with the restriction on communication between modules which allows messages to be sent only within a layer, or down one layer. This regularity, frequently employed for large systems, provides a useful framework within which a system can be constructed, managed and understood [19].

A single system may have many different regularities, each with its own specific role to play. As an example of a special purpose regularity, consider the following token-based protocol which might be employed by a distributed system S in order to ensure *mutual exclusion* with respect to a given operation O:

(a) No process performs operation O unless it possesses a certain token T.

(b) Initially, there is only one copy of T in the system.

(c) Token T may be transferred from one process to another, but no process ever duplicates T.

---

[2]We are using here the dictionary definition of the term regular as "conforming to a rule or principle".

Note that to be effective, this protocol must be a regularity; that is, it must be obeyed *everywhere* in the system, because the desired mutual exclusion would be endangered by any violation of this protocol, even by a single process.

Unfortunately, regularities are inherently hard to establish, unless they are imposed on the system by some kind of higher authority. The problem with regularities stems from their intrinsic globality. Unlike an algorithm or a data structure that can be built into few specific modules, a regularity is a principle that must be observed everywhere in the system, and cannot be localized by traditional methods. One can, of course, establish a desired regularity by painstakingly building all components of the system in accordance with it. But, as we shall see in Section 2, such a "manual" implementation of regularities is laborious, unreliable, unstable and difficult to verify and to change. While certain regularities are usually imposed by programming languages on all programs written in them, languages do not, and, as we shall argue later, probably cannot, support a sufficiently wide range of regularities.

We will explore here an approach to regularities which greatly simplifies their implementation, making them more easily employable for taming of the complexities of large systems. This approach, which is based on the concept of law-governed architecture (LGA) [13, 14], outlined in Section 3, provides system designers and builders with the means for establishing regularities simply by declaring them formally and explicitly as *the law of the system*. Once such a *law-governed regularity* is declared, it is enforced by the environment in which the system is developed.

Of course, not all desirable regularities can be thus established by means of formal laws -- for a law to be effective, it must be strictly and efficiently enforced. But the range of law-governed regularities that can be efficiently established under LGA is nevertheless quite broad. It includes, in particular, various module-interconnection frameworks, access-control schemes, a variety of inheritance structures, and certain protocols for distributed systems. A sample of these, and of some other kinds of feasible law-governed regularities is given in Section 4.

2. The Difficulties with Conventional Approaches to Regularities

Conceptually the simplest, and currently the most common, technique for establishing regularities is to implemented them *manually*; that is, to carefully construct the system according to the desired regularities. The problems with this approach, which are due to the inherent globality of regularities, are exemplified by the following difficulties one would have with the token-based regularity (or protocol) discussed in the introduction, if it is to be implemented manually:

1. It would be very *difficult to carry out* this implementation, because it must be done painstakingly in many different parts of the system. It would, in particular, be difficult to ensure that *no* process in the system `S` ever performs operation `O` without possessing the token `T`, and that no extra copy of `T` is ever made, anywhere in the system.
2. Any *verification* (formal or informal) that a given system satisfies this protocol, involves the analysis of *all* (or, at least, many) parts of the system.
3. This protocol would be very *unstable* with respect to the evolution of the system. Indeed, even if we are able to ascertain that the protocol is satisfied by a given version of the system, we cannot have much confidence in its satisfiability in future versions. Because, due to the global nature of the protocol, it can be compromised by a change *anywhere in the system*.
4. Finally, this protocol would be very *difficult to change*, even if the change itself is small, because such a change would have to be introduced *manually* into many parts of the system. Changes spread out in this way are very expensive and notoriously prone to error.

Since these problems are quite clearly endemic to all manually implemented regularities, it follows that it would be much better for regularities to be *imposed* on a system by some kind of "higher authority".

Perhaps the most obvious authority that can impose regularities is the programming language at hand. In fact, certain types of regularities are routinely imposed by various languages on the programs written in them. These include *block-structured* name scoping, *encapsulation*, *inheritance*, and various regularities involving *types*. In spite of the obvious importance of such built-in regularities, the imposition of regularities by means of a programming language has several serious limitations.

First, only very few types of regularities can be thus built into any given language; and a regularity built into the very fabric of a language tends to be rigid, and not easily adaptable to an applications at hand.

Second, programming languages usually adopt a *module-centered view* of software. They deal mostly with the internal structure of individual modules, and with the interface between a module and the rest of the system. But they generally provide no means for making explicit statements about the system as a whole, and thus no way for establishing regularities beyond what is built into the language itself.

A significant, but insufficient, deviation from this module-centered view of languages, is the *selective export* facility provided by some languages, like Eiffel, which allows one to specify in a given module the names of other modules that are allowed to send given messages to it. The limitation of the selective export facility as a mean for establishing regularities can be

demonstrated by its inability to support even such a simple regularity as layered structure. For this, as we shall see in the following section, one needs two capabilities: (a) the ability to organize all the modules of a system into groups; and (b) the ability to specify global constraints on the interaction between modules, based, among other things, on the user-specified grouping. None of these capabilities is generally provided by programming languages.

Finally, a language-imposed regularity is obviously not effective for multi-language systems, where regularities are particularly needed. For all these reasons it follows that the imposition of regularities requires a software architecture that provide a global, potentially multi-lingual, view of systems; such is our *law-governed architecture*.

3. An Overview of Law Governed Architecture (LGA)

The main novelty of law-governed architecture is that it associates with every system an *explicit* and *global* set of rules, which is *enforced* by the environment in which the system is developed. This set of rules is called the *law* of the system. The law is global, in that its rules have jurisdiction over the entire system. It thus provides the means of making statements about the system as a whole, statements which generally cannot be made with conventional programming languages. Unlike the *functional specification* of a system, the law is not merely a statement of intent, but, since it is strictly enforced, the law is an expression of *existing* regularities.

Generally speaking, the law deals with the internal structure of the system, not with its functionality. One can, for example, have a law that impose a layered structure on a system, quite independently of whatever it is that the system is doing. More specifically, the law governs the system under its jurisdiction by regulating the various interactions between its component-objects (or, modules) mostly ignoring the internal details of these objects. The interactions being regulated may be either *dynamic*, like the exchange of a message between pairs of objects, or *static*, like the existence of an inheritance relation between classes (in the case of an object-oriented language). The law, may, in particular, prescribe which objects can send which messages to which other objects; it may call for certain messages to be changed, or be rerouted to other than their nominal target; and it may prescribe some other actions to be carried out in conjunction with, or instead of, an attempted message.

The requirement that the law of a system must be strictly and efficiently enforced, means that we must restrict ourselves to very simple types of laws. Consequently, the law under LGA is, in particular, *differential* rather than *integral*. That is, the ruling of the law concerning a given interaction, depends only on the nature of the interaction itself, and on the immediate context of this interaction. Yet, although the law cannot deal explicitly with the effects that an interaction

may end up having on the system at large, a carefully designed law can have far reaching integral consequences, such as mutual-exclusion in the case of the simple token-protocol presented in Section 1.

Our concept of law-governed regularities can be fully understood and appreciated only in the broader context of LGA, which deals with the process of software development and evolution of software systems. To consider this context we now review very briefly the structure of Darwin, which is an experimental software-development environment that supports LGA[3] and we will present an example of a projects carried out under this environment.

The Darwin environment associates a law with every software-development *projects* managed by it. Besides governing the system under development, which is the focus of this paper, the law of the project governs the process of development and evolution of this system. Moreover, the law governs its own evolution throughout the lifetime of the project. The unified treatment of these seemingly distinct aspects of software and of software development, by means of a single concept of law, is unique to LGA; but it is quite necessary as we shall see via the example below.

A software development project starts under Darwin with the definition of the *initial law* that establishes the general framework within which the project is to operate and evolve. We will consider here an informal example of such an initial law, designed to support the development of *layered systems*. (For a formal expression of a similar law, see [14].

3.1. A Law of Layered Systems -- an Example

The example-law presented here partitions the modules of the system, developed under a project P governed by it, into groups called "layers". Using this grouping, the law imposes the well known *layered constraint* on the interaction between modules, as an *invariant of the evolution* of the system developed under P. Furthermore, this law provides for carefully circumscribed evolution of the law itself, allowing the manager of the project and its various programmers to establish certain kinds of additional regularities, throughout the course of software development. In particular, the manager is authorized to impose arbitrary *prohibitions* on the interaction between modules, while each programmer is authorized to specify which messages are *acceptable* to his own modules, subject to the layered constraint, and to the manager’s prohibitions.

This law is presented here, mostly informally, as consisting of four rules (enclosed in boxes)

---

[3]To be precise, this overview is of Darwin/2 [15], which is a major revision of the Darwin/1 environment described in [14].

that govern four distinct aspects of the project under its jurisdiction. Only rule r3 is specified formally, for the sake of illustration, but without detailed explanation of the formalism itself, which is described in [16].

Rule r1 below determines the *structure of the object-base* that would support project P throughout its evolutionary lifetime.

r1: The objects representing program-modules are partitioned into `layers`; the objects representing the builders of the system are partitioned into two roles: `manager` and `programmer`; and each module is designated as being *owned* by some programmer.

Technically, these partitions are defined by certain attributes associated with the various objects populating this project. The semantics of the resulting groupings of these objects is defined, in effect, by the other rules in this law, as we shall see.

Rule r2 governs the *process of development and evolution* under project P, by establishing the authority of the various builder-roles.

r2: A manager can create programmer-objects, and a *programmer* can make modules, becoming the *owner* of each module he makes. The owner of a module can program it, set its level, remove it, and pass its ownership to some other programmer.

This rule defines, in effect, what it means to be the *manager* of a project, and what it means to be an *owner* of a module. Note that the Darwin environment itself has no built-in concept of a "manager" or of "ownership", but, as we have just seen, it provides for such concepts to be established specifically for each project, by means of its law.

Rule r3 (which is specified *formally* below) governs the *structure of the system being developed*. It essentially states that a message `M` sent by `S` to `T` would be delivered to its destination only if the following three conditions are satisfied: (a) the layered constraint; (b) this message is not *prohibited* (as defined by `prohibited`-rules, which, according to rule r4 below, can be written into the law only by the manager); and (c) this message is *acceptable* to the target module `T` (as defined by the `acceptable`-rules, which, according to rule r4 can be written into the law only by the owner of `T`).

```
r3: sent(S,M,T) :-
      level(Ls)@S,level(Lt)@T,  /* The level of S and T are bound to  Ls and Lt, respectively.
      Lt≤Ls≤(Lt+1),              /* If this condition on levels is satisfied, */
      not prohibited(S,M,T),        /* and the message is not prohibited, */
      acceptable(S,M,T),            /* and it is acceptable to the receiver, */
      do(deliver(M)@T).                     /* then the message is delivered. */
```

Finally, rule r4 governs the *evolution of the law itself*, allowing the manager of the project, and the various programmers to refine the regularities of the system during its development.

r4: The law can change *only* as specified below:

1. A manager can add to the law (and remove from it) arbitrary `prohibited`-rules, which, according to r3, serve as prohibitions.
2. Every programmer can add to the law (and remove from it) `acceptable`-rules, which, according to r3, define which messages are acceptable to to any of *his own* modules, and which objects can send these messages.

Note that this particular law, which is, of course, merely an example of what can be done under LGA, establishes a framework which is analogous to, but much more general than, the conventional *module-interconnection frameworks* (MIFs) which are based on *selective export*, as in Eiffel in particular. The analogy comes from an apparent similarity between Eiffel's *export statements*, and our `acceptable`-rules. Both are anchored on a module, defining the kind of messages that can be sent to it.[4] Yet, our example-law has several characteristics which are unmatched by Eiffel, or by any conventional MIF known to the author.

First, under this law the layered-constraint is an *invariant of the evolution of this project* -- unchangeable even by its manager. Our ability to establish such invariants of evolution, without them being hard-wired into the environment, or into the language at hand, can attest to the power of this architecture.

Second, using `prohibition`-rules the manager can establish additional regularities, over the entire system. For example, the manager may prohibit modules designated as *tested* from calling untested ones.

Finally, even our `acceptable`-rules are significantly more general than the export-statements. The Eiffel's export-statements in a given module `m` list explicitly the *names* of the modules that can send a given message to `m` (unless it is a universal export). In our case, on the other hand, the analogous specification, by means of the `acceptable`-rules, can be by some condition defined over the attributes of the various modules of the system. These allows programmers to formulate *general prescriptive policies*, concerning the use of their modules. Here are two, informally stated, examples of policies concerning a given module `m` that can be expressed by means of a single `acceptable`-rule writable by the programmer of `m`.

- Module `m` can accept a specified message from *every module at the layer of* `m`.

[4]Because while the export-statement must be included textually within a module, our `acceptable`-rules are writable by the owner of the module they are anchored on -- not a major difference.

- Module `m` can accept a specified message from *every module owned by a programmer Jones.*

3.2. On the Implementation of Darwin

The Darwin environment has essentially two levels: (1) the *abstract*, language-independent, level that maintains an object-base of the project, defines the concept of law, and provides a framework for its enforcement; and (2) the *concrete* level that contains a set of *language interfaces*, one for each programming language which may be used by any of the modules of the system. Currently, Darwin has interfaces for two, very different, languages: an interface called LGA/Prolog, for the logic-programming language Prolog [4]; and an interface called LGA/Eiffel for the object-oriented language Eiffel [8]; and an interface for C++ is being planned.

A *language-interface*, for a given language, performs two functions. First, it maps the abstract concepts of this architecture, such as *object* and *interaction* between objects, to certain concrete constructs of the language at hand. For example, our Eiffel-interface maps the abstract concept of objects in Darwin to Eiffel-classes, and it maps the abstract concept of interaction to procedure-calls that cross class-boundary, and to various static relationships between classes, such as inheritance and redefinition. The second function of a language-interface in Darwin is to provide the language specific part of the law-enforcer.

Space limitation rules out any detailed discussion of Darwin's mechanism for law enforcement. We will say here only this: The laws defining the various regularities discussed in this paper can be enforced either purely at compile-time, without any run-time overhead, or with minimal run-time overhead due to certain code inserted at compile time in various places in the program.

3.3. Related Work

The idea that a large system needs to be governed by an *explicit*, *global* and *enforced* set of rules (which we call "law") is not entirely new. It appeared in several incarnations in various kinds of systems, but so far it has never been developed into a full fledged architecture for general software systems, of the kind described here. The following are the main such efforts known to the author:

- Perhaps the earliest attempt to provide an explicit global law for general software systems (not for some specific kind of systems, like databases) was by Ossher [18, 19]. He built a mechanism that imposes a specified *layered* module-interconnection structure on a given system, which bears some similarity to our example-law discussed in this section.
- The Meta system of Marzullo and Wood [7] has a global set of "policy rules" about the

interaction between nodes in a distributed system. Like our law, these rules are explicit and enforced, and they serve as a kind of "glue" to bind the various pieces of the system together.This system deals exclusively with certain aspects of distributed system.

- Finally, and most recently, Shoham and Tennenholtz [23], using a rationale very similar to our own, proposed the use of global laws to regulate distributed systems of robots. They discuss techniques for determining the appropriate laws (they actually call them "laws") in certain situations, but so far they proposed no mechanism or general architecture to enforce such laws.

4. A Sample of Law-Governed Regularities

In Section 3 we discussed a particular kind of law-governed regularities, involving constraints on the exchange of messages between the modules of a system. In this section we discuss several additional kinds of useful regularities that can, and have been, established by laws under LGA. The regularities to be discussed here are not new *per se*. Some of them have been incorporated into certain conventional languages and other computational platforms, others have been formulated by system designers and programming managers, and have been implemented "manually" into various systems, with all the disadvantages of such an implementation. What is new here is that under LGA, all these different regularities can be formulated explicitly as laws, and then efficiently enforced by the environment in which the system is developed. Such regularities are easier to establish and to change, and are far more reliable then the manually implemented ones.

4.1. Regulating the Use of the Unsafe Primitives of a Language

Every practical programming language has some *unsafe* primitive features whose careless use may have disastrous consequences. The use of many of these features can be carefully regulated under LGA, helping to make systems more reliable, safer, and in a sense simpler.

A familiar example of an unsafe feature of a language is the `dispose` primitive of Pascal, which, if used carelessly, may cause the phenomenon of *dangling reference* with its quite unpredictable consequences on the entire system. Additional examples of unsafe features which are common in languages include *address arithmetic*, and certain *type conversions*. Careless use of any of these features, and of others like them, can violate the semantics of the host language. In particular, the many unsafe features of C++ may allow any object to manipulate the private space of other objects, thus violating the principle of *encapsulation* -- perhaps the most important regularity of object-oriented programming.

Certain features of a language may be considered unsafe even if they do not violate the semantics of the language. For example, in a patient-monitoring system, access to the actuators that control the flow of various fluids and gases into the body of a patient is clearly unsafe, as it is

crucial to the life of this patient. More generally, in *embedded systems*, the ability to operate on the outside world (typically represented by system-calls) is often similarly unsafe.

The worth of regulation over the use of unsafe primitives can be demonstrated by the kernel-based architecture of operating systems. The *kernel* is a small part of the system which presents the rest of it with several fundamental abstractions such as that of a *process* and of *virtual-memory*, which, if the kernel is written correctly, are invariant of whatever is done outside the kernel. This architecture is made possible by the *confinement* to the kernel of certain unsafe capabilities provided by the bare machine, such as the ability to interrupt the CPU, and the unrestricted access to the main memory. This confinement to the kernel is a regularity, in our sense of this term, because it is a statement about the entire system. Namely that no part of the system except the kernel can use the unsafe primitives in question.

The need for such confinement is not restricted to operating systems, of course. It would, for example, be invaluable for the above mentioned patient-monitoring system, which can be made much safer if access to the various critical actuators is *confined* to few specific modules that are supposed to manage them. And C++ programs, in general, can be made much more reliable, and easier to debug, if the various unsafe features of the this language will be confined to the regions of the program that really need them, while the rest of the system is subject to the much stricter object-oriented discipline.

Unfortunately, with few notable exceptions, programming languages generally do not provide any means for regulating the use of the unsafe primitives built into the language itself. Note that the special hardware that supports confinement in operating systems is not available for use inside user-programs outside the kernel.

One of the few languages that does allow for some regulation over its unsafe features is Modula-3 [3]. This language explicitly characterizes certain of its primitive features as unsafe, allowing them to be used only inside modules that are explicitly declared to be "unsafe". This is a step in the right direction, but it does not go far enough. It should, in particular, be possible to regulate the various unsafe features of a language *individually*, by, for instance, confining each to a different module. It should also be useful to regulate the interaction of such "unsafe modules" with the rest of the system, and to regulate the construction and evolution of such modules. Such flexible controls, and others, are possible under LGA.

Under LGA/Eiffel, in particular, the law governs the ability of an Eiffel-class to have procedures written in C -- the main unsafe feature of the Eiffel language. Thus, one can confine the use of C to certain classes, or to certain types of classes, say, classes written by a certain group of

programmers, or classes residing in the lowest layer of the system. Moreover, one can regulate the interaction of classes that use C with the rest of the system. For example, only certain classes may be allowed to inherit from classes that use C. Many other unsafe features would be subject to control under the planned LGA/C++ interface.

4.2. Token-based Regularities

Many useful structures and control mechanisms can be based on the notion of a *token* -- an item that, by its dictionary definition [1], "tangibly signifies authority and authenticity". One example is the mutual exclusion protocol discussed in the introduction, where the movable but unique token T represents the exclusive authority to perform operation O. Another example is the well known *capability-based* access control mechanism [5], under which operations on objects are authorized by tokens called "capabilities". There are also many real-life processes which are regulated by tokens. For example, entry into theaters is regulated by means of tokens called *theater-tickets*; much of the financial activities of our society are largely regulated by tokens called *dollars*; and access to roads is regulated by means of tokens called "tokens". These and other real-life token-controlled processes have close analogies in computer systems, and are themselves often subject to computer support.

The great virtue of token-based control is that the validity of an operation can be determined strictly *locally*, on the basis of the token being presented. But this locality depends on the existence of some underlying global regularities. Specifically, for an item T to serve as a token in a given system, conferring on its holder the power to perform a certain operation O, the system must satisfy two kinds of regularities: First, it must be *impossible* to perform O without possessing T; and second, the *creation and distribution of T-items must be strictly controlled*, so that the mere possession of a token can be taken as *prima facie* proof of the authority it is supposed to represent. As has already been explained, unless these two regularities are imposed on the system by some higher authority, they are very difficult to establish reliably. This is particularly true for distributed systems where the management of tokens cannot be centralized. (We note here that while cryptographic techniques can be used in many cases to ensure the *authenticity* of tokens, they are not very helpful in providing tokens the *exclusive authority* which make them meaningful.)

Most programming languages give no support for token-based regularities, since languages generally provide for almost no control over the creation of objects, and over the transfer of objects (or of pointers to objects) from one part of a program to another. While some specific token-based regularities have been built into certain computational platforms (like operating system [5]), and into some programming languages [24], none of them provides the means for establishing a broad

spectrum of such regularities, of the kind described below. Consequently, if a certain token-based regularity is desired in a given system, it would usually have to be implemented manually, with all the disadvantages of such an implementation.

Under LGA, on the other hand, it is possible to establish a wide range of token-based regularities simply by means of appropriate laws. The range of regularities that can be supported under LGA span several dimensions, such as:

- *The nature of the authority represented by a token.* In particular, a token may represent the right, or power, to operate in a certain way on a single object, as in the case of a *capability*; or, in some analogy to a master-key, a token may provide the power to operate on a whole set of objects.
- *The manner in which a token itself is affected by the operation which it is used to authorize.* In particular, the token may be completely unaffected by its use, and thus be usable an indefinite number of times; it may be usable just once; or some specified number of times.
- *The controlled means provided for the creation of tokens, and for their distribution.* For example, the ability to move a token from one object to another may be conditioned on the availability of certain other tokens, as proposed in [6, 9]. Also, tokens may be allowed to be copied, or just be moved from one place to another.

Several examples of laws that impose various types of token-based regularities under LGA are discussed in [13] and in [16].

4.3. Monitoring

There are many situations in which one would like to monitor a certain kind of messages by notifying a given object (the "monitor") of their occurrence. Such monitoring can be useful for debugging; it may help in fine tuning a system by collecting statistics about various interactions between its parts; it may be used to provide various parts of a system with information about the activity of other parts, as it is done in the GARDEN system [20]; it may help in the defense against viruses and other attacks on a system; and it may facilitate on-line auditing of financial systems.

A specific monitoring regime is often a regularity, in the sense that it requires a certain protocol to be observed by many different parts of the system, and it is, therefore, difficult to implement manually. Moreover, there is a wide range of possible monitoring regimes, not all of which are likely be built into any one programming language or environment. In particular, monitoring regimes may differ along the following dimensions: (a) the nature of messages to be monitored, and the circumstances under which they should be monitored; (b) the identity of the monitor; (c) the actions that should be carried out when a message to be monitored occurs; and (d) the mechanisms for starting and stopping a monitoring activity. Many such regimes can be effectively established by law under LGA, and several of them have been experimentally implemented under Darwin.

4.4. Regularities in Object-Oriented Systems

In a series of previous papers [10, 22, 11, 12] we have shown that most of the fundamental structures of object-oriented programming, including *inheritance* and *delegation*, and many variations of these structures, are regularities that can be established under LGA by means of explicit laws. While the resulting flexibility should be highly beneficial for exploratory programming, most *object-oriented systems* are likely to be built in conventional object-oriented languages, such as C++ or Eiffel, which support some fixed concepts of inheritance, and of related structures. In this section we discuss some of the law-governed regularities which are likely to be useful for systems written in one of these conventional object-oriented languages.

First, the various types of regularities discussed so far in this paper are applicable to OO-systems, just as to any other kind of systems. In particular, it is often useful to group the classes of an OO-system into various clusters, by associating appropriate attributes with the objects that represents these classes in the object-base of Darwin. Such a partition can then serve as a basis for various regularities, as we have done for layered systems in Section 3. Also, the regulation of unsafe features may be very beneficial for OO-systems, particularly for systems written in C++, which has a lot of very unsafe features to regulate. Token-based and monitoring regularities should also be useful for OO-systems.

In addition, the special structures of OO-systems require special regulations. In particular, one may want to regulate which classes *can*, or *must*, inherit from which other classes. Also, one may want to establish the permissible relationships between a parent and its heirs, e.g., which aspects of the parent can be *visible* to, *redefined* by, or renamed by which of its heirs. For farther discussion of such regularities, and their motivation, the reader is referred to [17], which is based on our work with the LGA/Eiffel interface. In the following sub-section we present a detailed example that motivates some of these, and some other regularities in OO-systems.

4.4.1. An Example: Creating a Killable Class of Objects

Consider a system built in an OO-language such as Eiffel, that provides garbage collection. It is well known that in such a language it is impossible to explicitly remove an object, because there may be references to it anywhere in the system. Given this, suppose that we would like to create a class C of objects that are *killable* in the following sense. A message `kill` sent to a C-object `x` should have the effect specified below (where by the term C-object we mean an instance of class C, or of any class that inherits directly or indirectly from C.)

1. A certain "burial procedure" B should be carried out on `x`.
2. Any subsequent attempt to send any message to the "killed" object `x`, *from anywhere* in the system, should result in a certain *error response* E.

Now, it turns out that it is impossible to ensure such behavior for all C-objects, without building it manually in many parts of the system. About the best we can do in Eiffel is the following:

(a) Build into class C a boolean attribute `alive` whose initial value is true, and a method `kill`, programmed to set the attribute `alive` to false, and to perform the required burial procedure B.

(b) Make sure that the method `kill` cannot be redefined by *any* class that inherits (directly or indirectly) from C.

(c) Have *all* methods of class C, and of *any* direct or indirect heir of C perform the error response, E if applied to an object that has been killed.

Part (a) of this implementation is localized, in the class C, and is thus easy to accomplish. The other two parts, however, are regularities which must be satisfied in many parts of the system. It so happens that the regularity (b) above is supported by Eiffel, which allows a feature of a class to be declared as *frozen*, preventing it from being redefined by any heir of this class. But regularity (c) must be carried out manually, by programming in the specified manner *all* methods of *all* the classes that inherit directly or indirectly from C -- a potentially difficult, and highly unreliable proposition. Thus, Eiffel provides us with no assistance in establishing this kind of regularity, nor does any other object-oriented language. Under LGA/Eiffel, on the other hand, the problematic regularity (c) above can be easily imposed by the law of the system.

5. Conclusion

In his book Symmetries and Reflections, the theoretical Physicist Eugene Wigner wrote [25]:

> "Physics does not endeavor to explain nature. In fact, the great success of physics is due to a restriction of its objectives: it only endeavors to explain the *regularities* in the behavior of objects".

Software engineering should, perhaps, similarly focus on the formalization of regularities, and on understanding their role in software systems. This paper is a step in that direction.

Bibliography

[1] Morris, W.
The American Heritage Dictionary of the English Language.
Houghton Mifflin Company, 1981.

[2] Brooks, Frederick P. Jr.
No Silver Bullet -- the Essence and Accidents of Software Engineering.
IEEEComputer :10-19, April, 1987.

[3] Cardelli, L. Dinahue, J. Glassman, L. Jordan, M. Kalsow, B. and Nelson, G.
Modula-3 Report (revised).
Technical Report 52, Digital System Research Center, November, 1989.

[4] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[5] Denning, P.J.
Fault tolerant operating systems.
Computing Surveys 8(4):359-389, December, 1976.

[6] Harrison, M. A., Ruzzo, W. L. and Ullman, J. D.
Protection in operating systems.
Communications of the ACM 19(8):461-471, Aug., 1976.

[7] Keith Marzullo and Mark D. Wood.
Tools for Monitoring and Controlling Distributed Applications.
Technical Report TR91-1187, Cornell University Department of Computer Science, February, 1991.

[8] Meyer, B.
Object-Oriented Software Construction.
Prentice-Hall, 1987.

[9] Minsky, N.H.
Selective and Locally Controlled Transport of Privileges.
ACM Transactions on Programming Languages and Systems (TOPLAS) 6(4):573-602, October, 1984.

[10] Minsky, N.H. and Rozenshtein, D.
Law-Based Approach to Object-Oriented Programming.
In *Proceedings of the OOPSLA'87 Conference*, pages 482-493. October, 1987.

[11] Rozenshtein, D. and Minsky, N.H.
Law-Governed Object-Oriented System.
Journal of Object-Oriented Programming 1(6):14-29, March/April, 1989.

[12] Minsky, N.H. and Rozenshtein, D.
Controllable Delegation: An Exercise in Law-Governed Systems.
In *Proceedings of the OOPSLA'89 Conference*, pages 371-380. October, 1989.

[13] Minsky, N.H.
The Imposition of Protocols Over Open Distributed Systems.
IEEE Transactions on Software Engineering , February, 1991.

[14] Minsky, N.H.
Law-Governed Systems.
The IEE Software Engineering Journal , September, 1991.
(This is a revision of a similarly entitled 1987 technical report).

[15] Minsky, N.H. and Rozenshtein, D.
Specifications of the Darwin/2 Environment.
Technical Report, Rutgers University, LCSR, 1991.

[16] Minsky, N.H.
Regularities in Large Systems.
Technical Report, Rutgers University, LCSR, 1993.
(In preparation).

[17] Minsky, N.H. and Pal, P.
Imposing Regularities over Object-Oriented Systems.
Technical Report, Rutgers University, LCSR, 1993.
(In preparation).

[18] Ossher, H.L.
Grids: A New Program Structuring Mechanism Based on Layered Graphs.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 11-22. January, 1984.

[19] Harold L. Ossher.
A Mechanism for Specifying the Structure of Large, Layered Systems.
In Bruce Shriver and Peter Wegner (editor), *Research Directions in Object-Oriented Programming*, pages 219--252. MIT Press, 1987.

[20] Reiss, S.P.
Working on the Garden Environment for Conceptual Programming.
IEEE Software 6(4):16-27, November, 1987.

[21] Robinson, L., Levitt, K.L., Neuman, P.G. and Saxena, A.R.
On Attaining Reliable Software for a Secure Operating System.
In *Proceedings of the 1975 International Conference on Reliable Software*, pages 267-264. IEEE, April, 1975.

[22] Rozenshtein, D. and Minsky, N.H.
Constraining Interactions between Objects in the Presence of Class Inheritance.
In *Proceedings of the 2nd International Workshop on Computer-Aided Software Engineering*. July, 1988.

[23] Yoav Shoham and Moshe Tennenholz.
On the synthesis of useful social laws for artificial agents societies (preliminary report).
In *Proceedings of AAAI-92*. 1992.

[24] Strom, R.E.
Mechanism for Compile-Time Enforcement of Security.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 276-284. January, 1983.

[25] Wigner, E. P.
Symmetries and Reflections.
Ox Bow Press, 1979.

Table of Contents