# DARWIN-E: AN ENVIRONMENT FOR IMPOSING REGULARITY ON OBJECT-ORIENTED SOFTWARE

**BY PARTHA PRATIM PAL**

**A dissertation submitted to the**

**Graduate School—New Brunswick**

**Rutgers, The State University of New Jersey**

**in partial fulfillment of the requirements**

**for the degree of**

**Doctor of Philosophy**

**Graduate Program in Computer Science**

**Written under the direction of**

**Naftaly H. Minsky**

**and approved by**

_____

_____

_____

_____

**New Brunswick, New Jersey**

**October, 1996**

## ABSTRACT OF THE DISSERTATION

# Darwin-E: An Environment For Imposing Regularity on Object-Oriented Software

**by Partha pratim Pal**

**Dissertation Director: Naftaly H. Minsky**

*Regularity* in this dissertation means conformity to unifying principles – principles that affect every part of a software system, or, some significant and well-defined subset of its parts. Complexity of a large system can be reduced by incorporating some meaningful regularity into it. Regularity can be used to explicate structural aspects that were invisible otherwise. Furthermore, various thumb-rules of software construction, certain architectural patterns, and constraints underlying various design models can also be expressed as regularity.

An explicit and formal statement of the desired regularities therefore, can serve as a guideline to construct a well-designed software that is easier to understand and maintain. Strict imposition of these regularities on the other hand leads to a controlled evolution of the software, with a guaranteed conformance to its desired model and underlying guidelines.

Object-orientation is the technology of choice today for building large systems. We considered software development in Eiffel [37], one of the advanced object-oriented languages and showed that there are many cases where regularities would increase the comprehensibility, mangeability and reliability of Eiffel systems [48]. However, neither the language nor its host environments has the infrastructure for formulating

and imposing such regularities.

The paradigm of *Law-Governed Architecture* (LGA) [43] provides the perspective needed for realizing regularities: regularities can be formulated and enforced as the "law" under LGA. We developed a prototype environment named Darwin-E that supports LGA for Eiffel. The novelty of the Darwin-E environment is its ability to perform compile-time enforcement of a wide range of useful laws, thus imposing desired regularities upon object-oriented (Eiffel) systems being developed in it. Several other aspects of a software project that benefit from LGA, such as controlling developer activities, configuration management and monitoring, are also handled in Darwin-E.

The main contribution of this dissertation is that it demonstrates the feasibility of using regularities as an integral part of object-oriented software construction. Besides, being an environment whose principal focus is the software product, Darwin-E complements the approaches taken by process-centered environments.

# Preface

The difference between a small system and a large system is that the latter cannot just be *built*, it has to be *engineered*. Engineering sciences rely on physical laws of nature (such as the laws of mechanics or the laws of thermodynamics) that dictate the structure of the system and regulate the complex behavior of its components. Software engineering, the discipline that aims to tackle the *engineering* problems of large software systems and their production process, unfortunately, lacks the support of natural laws: there is no natural law that governs the behavior of modules of code or the human programmers that create and modify them. Therefore, artificial means like social laws are utilized in the form of language rules, protocols, methodologies, conventions and guidelines. While language rules and protocols can be enforced by the compiler and other tools, software systems in general lack the infrastructure to deal with the laws, their legislation, their enforcement and their evolution. The Law-Governed Architecture(LGA) was developed as an attempt to provide such an infrastructure. In this dissertation we investigate object-oriented software under LGA. Our objective is to formulate and impose by law certain regularities on the target software in a manner that is invariant of the evolution of the project. The Darwin-E environment described here demonstrates that it is possible to do so.

# Acknowledgements

I would like to thank my parents who brought me into this world in the first place and introduced me to the universe of books and computers, my advisor Naftaly Minsky for his patience, support, help and guidance, the members of my thesis committee B. R. Badrinath, Alex Borgida and Naser Barghouti, for their helpful insights and suggestions to improve the work, and my innumerable friends and colleagues at Rutgers and elsewhere, without whose support and love, life would have been difficult.

Special thanks go to Valentine Rolfe, for her kindness throughout my stay here at Rutgers and to my wife Mou, for patiently and cheerfully sharing the life of a graduate student in a foreign country.

I would also like to thank my brother, with whom I shared my childhood and my mother, who has always been the best source of inspiration and happiness.

# Dedication

The person who would be the happiest to look at this work, left this world in May 1991, even before it was started. I dedicate this dissertation to the fond memory of my late father.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Problems With Large Software Systems

Large software systems, like any other large systems, cannot just be *built*, they have to be *engineered*. However, there are several problems that are unique to software systems, and do not arise in other engineering disciplines. Brooks identified *complexity* and *invisibility* as the principal ones amongst them [7]. He noted that *complexity* is an inherent and irreducible property of large software. He also noted that as opposed to any other kind of product, the details and intricacies of a large software product by nature are *invisible*. The problem is further aggravated by the extreme *malleability* of computer programs and the *freedom* and *uncontrolled power* that a programmer enjoys while creating a computer program as opposed to, say, a civil engineer constructing a bridge. Therefore, it is very hard to understand and reason about the structure and design of large software systems. People do attempt to rationalize the structure of a system according to rules, but it is very easy for the programmers to violate the rationale and underlying principles of the high-level design while realizing it in their code. The primary reason is that the global perspective of such a system is beyond the comprehension of a single programmer. Furthermore, the rationale and principles are normally not explicit parts of the software, and there is no infrastructure, such as the omnipresent physical laws of nature, to guide and constrain the way programmers put together their code and make them work. A corollary of this is that during the evolution of software systems, it is very natural to deviate from the original design, resulting in incomprehensible and unmaintainable systems. These problems are already serious enough, and because software systems are becoming increasingly larger, will continue

to aggravate.

## 1.2 The Notion of Regularity

In this dissertation, the term *regularity* refers to any *global* property of a system; that is, a property that holds true for every part of the system, or for some significant and well defined subset of its parts. Thus the statement *class B inherits from class C*, in an object-oriented system, does not express a regularity, since it concerns just two specific classes; but the statement **every** *class in the system inherits from C* does express a regularity, and so does **only** *class B inherits from class C*, both of which employ universal quantification.

Regularity should not be confused with specification. The example regularities presented above have no place in the users requirements as they deal with the inside of the system, and statements such as those in the specification of a system would be considered gross over-specification. Specification defines the interface of a module and its responsibility. However, a system is not just a collection of modules, it also involves some *unifying principles*, some regularities, which are meant to make the system simpler to understand, easier to build and maintain. Such regularities may either be implicit in the underlying design model or may have been chosen on purpose in a conscious effort to achieve a certain goal.

As an illustration, consider a project developing *kernelized* software with the following design model. There are two distinct clusters of classes, namely the kernel cluster and the application cluster. The kernel cluster consists of a small set of classes that wrap around the hardware and present a tamed and safe abstraction to the application cluster. That the kernel classes are the only classes who can directly access the hardware features, is one of the underlying principles of this design model, which means *any* kernel class can access hardware, but *none* in the application cluster can. Therefore, they must call kernel classes to use hardware services. This *global* property of a kernelized system, which qualifies to be a regularity, does not relate to the specification of the particular system at hand, rather it reflects the very essence of the design model.

If a software product is known to possess the above example regularity, we can clearly visualize the kernelized structure of the system: critical hardware accessing functions localized within the small kernel and application classes going through the kernel, in order to use hardware features. This makes the system easy to comprehend and maintain, and more reliable than a system in which any class can directly access the hardware.

## 1.3 Regularity In Large Software

A system with some regularity is less complex than a system without any regularity. The same is true for software systems. In [45], Minsky observed that large systems are inherently incomprehensible if they do not posses some regularity. At the same time, regularity can bring out some of the hidden aspects of the software system, reducing the inherent invisibility. Various thumb-rules of software construction that are known to yield quality software, different architectural patterns that the experts frequently use as idioms, and the constraints and principles underlying various design models can also be formulated as regularities. An explicit and formal statement of the desired regularities therefore, can serve as a guideline to construct a well-designed software that is easier to understand and maintain. Strict imposition of these regularities on the other hand leads to guaranteed conformance to its desired model and underlying principles. This, aided by a regulated software development process, can result in a *controlled evolution* of the software product, immune to much of the problems arising out of extreme malleability of software and unconstrained power of the developers.

Of-course, we do not intend to claim that regularity is the *silver bullet*. Regularity alone cannot make all large systems simple, nor can all kinds of useful software engineering constraints be formulated and enforced as regularities. Brooks [7] and other researchers [19] have already established that there is no such cure-all solution. What we provide is an engineering solution: the designers of a large system are to come up with a set of regularities in a conscious effort to fight some of the problems of large software systems. We provide a means to impose such regularities.

## 1.4 Support for Regularity in Existing Technology

The simplest and currently the most common technique for establishing regularities is to implement them *manually*, by painstakingly building the system according to the desired regularities, which are normally posed as policies, principles, design rules, guidelines or conventions. Programming languages and tools provide very little support for these and it is generally assumed that they will be followed by the programmers.

Regularities are hard to implement this way, because of 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 thus cannot be localized by traditional methods. Besides, such a manual approach is *laborious, unreliable* and *difficult to verify* and can be compromised by a change anywhere in the system. It is hard to escape the conclusion that to the extent possible, regularities should be *imposed* by some kind of an "higher authority".

The obvious candidate for such a higher authority is the programming language in which a system is written. While certain types of regularities, such as *block-structure*, *encapsulation*, and *inheritance*, are usually imposed by programming languages, this approach of incorporating regularities as built-in language facilities has several serious limitations. First, a regularity that is built into a language tends to be rigid, and not easily adaptable to the needs of an application at hand, whereas the regularities we are considering do not have an universal applicability. Second, regularities often involve various project-specific or application-specific aspects (for instance layer number or ownership of a class or the status of the owner of a class) that are beyond the scope of general purpose programming languages. Finally, programming languages usually adopt a *module-centered view* of software. They deal mostly with the internal structure of individual modules, and with the interface of a module. Languages generally provide no means for making explicit statements about the system as a whole, and thus no means for specifying global constraints over the interactions between the modules of the system, beyond the constraints built into the language itself. It follows that the infrastructure for our regularities should be something other than programming

languages.

## 1.5 Regularity And LGA

The Law-governed Architecture (henceforth referred to as LGA) [43] provides a framework for formulating and enforcing artificial laws about software systems in a manner similar to the laws of a society that govern our societal behavior. It associates with every software development project an *explicit* set of rules, which is *enforced* by the environment that manages the project. The set of rules is collectively known as the *law* of the project. The *law* provides a unified paradigm of control over two seemingly disparate aspects of a software project: the software product and its development process. The *law* is *global*, that is, its rules have jurisdiction over the entire project. It thus provides the infrastructure needed for our regularities that generally is not found in conventional programming languages and software development tools.

## 1.6 Regularity in Object-Oriented Software under LGA: The Darwin-E Environment

It is generally accepted that object-oriented design and methodologies are suitable for large systems and accordingly, most of today's large software systems will be implemented in some object-oriented language. But, even state of the art object-oriented languages such as Eiffel, and their host environments (such as the ISE Eiffel 3.0 environment or the TowerEiffel environment [1]) are not equipped to support the regularities that are quite essential for large software systems. We have chosen the Eiffel language for this research, because it is a pure object-oriented language with a clean design. To illustrate the use of regularities in the software engineering of Eiffel systems, we created the Darwin-E environment. This *proof-of-the-concept prototype* is a major part of this dissertation.

---

[1] ISE Eiffel3.0 is a commercial product developed by Interactive Software Engineering Inc. and TowerEiffel is another commercial product developed by Tower Technologies Inc.

Darwin-E is a software development environment (for Eiffel), where one can define for each project, a customized set of rules known as the *law* of the project, to *regulate the software development process* and to *impose a certain regularity on the software being developed.* These rules lead to a *controlled evolution* of the project, since these rules are strictly enforced during the life of the project. The rules need not remain unchanged throughout the life of the project, and the law itself *controls changes made to the law.* In this dissertation, we will describe the Darwin-E environment and use it as a test-bed for various useful regularities.

Although Darwin-E provides some control over the activities of human participants in the software project, it is *different* from the traditional process centered environments like Marvel [2, 3, 24], Merlin [23] or Adele/TEMPO [4]. The main goal of these process centered environments is to enact and automate a software process defined in some formal language, which is not pursued by Darwin-E; in contrast regularity of the product is our primary focus, which is often not considered in the process centered environments. However, some recently proposed software development tools and environments such as the *Reflexion Model* [54], pattern-lint [65] or the Demeter tools [34] do focus on the design and architectural model of the software product much like Darwin-E.

Darwin-E itself is a fairly big and complicated system under LGA, that was built incrementally. Apart from the software engineering lessons learnt during the design and development of this system, it demonstrates that LGA can be applied to different languages belonging to different paradigms. In fact, by changing the language front-end for Darwin-E, similar environment for other languages can easily be built.

## 1.7 Contribution of This Dissertation

The main contribution of this dissertation is that it establishes the *usefulness* and *usability* of regularities in software engineering. We show that regularity helps in reducing the complexity of object-oriented (Eiffel) software by imposing a desired and explicitly visible structure. We also show that our mechanism provides a basis for making software systems conform to the underlying principles and architectural patterns of the

original design in their implementation and evolution.

Among the numerous useful regularities that we have investigated are the notions of *immutable objects*, *side-effect-free functions* and various levels of *access-control by intention*. We examined well-known architectures for large systems such as *layered* or *kernlized* design and show how to formulate and enforce the principles underlying them. We established that the recently popularized *design patterns* [16, 59] which lack the enforcement support, have inherent implicit constraints that can be formulated as our regularities and hence, can be enforced. We also show that well-known thumb-rules that are often used in object-oriented design and implementation such as *The Law of Demeter* [31] can also be enforced in a project-specific manner as a regularity.

Darwin-E supports LGA , which is the basis of our regularities. Among the other benefits derived from the law-governed framework in Darwin-E, are its configuration management and monitoring facilities. Darwin-E features a novel consensus based configuration binding facility that considers the interests of different concerned parties in the configuration binding process. The monitoring facilities provide for watching over developer activities in a project, as well as code instrumentation for on-line monitoring.

The Darwin-E environment itself is significant in another way. It is a product-centered environment. We would like to pose Darwin-E and our approach to impose regularity on software product as a demonstration of the potential of product-centered methodologies and environments in software engineering research and practice. The merit of such approach is corroborated by the recent surge of product-centric research as documented in [16, 17, 34, 54, 65].

# Chapter 2

# Background: A Brief Discourse On Law-Governed Architecture

Law-Governed Architecture (LGA) serves as the basis on which support for regularities are implemented. A brief discourse on LGA, is therefore necessary not only to provide the background required for understanding this dissertation, but also to make it self-contained. This chapter is a composition of materials extracted from various LGA papers, such as [43, 44, 45].

## 2.1 Law-Governed Architecture and Its Realization in Darwin

LGA provides an integrated framework for formulating and enforcing rules about the software being developed and its development process. Every software project $\mathcal{P}$ under LGA (by "project" we mean an evolving system, together with the team of programmers that maintains it) is associated with a set $\mathcal{L}$ of explicitly-defined rules, known as the *law* of the project, that is strictly enforced by the environment that manages $\mathcal{P}$. $\mathcal{P}$ can be described in an abstract manner as a set $\mathcal{O}$ of interacting objects, where the inter-object interactions are subject to $\mathcal{L}$. This abstract LGA model is implemented in the Darwin environment and we will often describe LGA concepts in terms of its realization in Darwin. We begin with a discussion of the objects under LGA.

### 2.1.1 LGA Objects

An object under LGA is an abstraction of the various entities involved in a project, ranging from software components to human users. In the abstract model, an object is conceived as a pair, (*agent, state*), where the agent is the active component and the

state is a passive repository of data structures.

The agent is responsible for driving its host object. There can be three kinds of agents, and by implication, three kinds of objects:

- A *programmed agent*, which is driven by its program text (also known as its *script*).
- An *un-programmed agent*, which is driven by human users (usually an unpredictable initiator of inter-object interaction).
- An *idle* agent, which does not initiate any activity by itself (but may be involved in an activity initiated by others).

The programmed objects model software components in $\mathcal{P}$; they may represent software tools that are used in $\mathcal{P}$ or modules of a software system being developed in $\mathcal{P}$. The un-programmed objects model human components in $\mathcal{P}$ and may serve as the loci of activities of a human user in a project. The idle objects model the database components of $\mathcal{P}$ and may represent passive data structures such as the collection of different versions of a module, as required for managing $\mathcal{P}$. In the abstract model we make no commitment about these.

The activities of an agent include interacting with other objects and executing operations within itself. For instance, a script of a programmed object may invoke an operation on another object or may evaluate a function in its own script without interacting with anyone. The later is not subject to the law and the internals of a programmed agent (for instance, its runtime state) is treated as a black box.

The state of an object is *accessible* to the law, which is to say that *the law can make distinctions between objects on the basis of their state and changes in the state are regulated by the law.* It stores various properties (also called attributes) associated with the object. In general, the properties are Prolog [9] terms of the form `functor(arg1, arg2,..)`, where arguments `arg1, arg2,..` can be literals, Prolog terms or variables. With few exceptions (like the property `#id(i)` which stores the identity of the object `i`), LGA does not assign any semantics to the attributes of an object; it is the law of the given project that gives it meaning.

### 2.1.2 LGA Interactions

The interaction between LGA objects can be categorized as:

1. Dynamic operations carried out by one object on another, such as operations invoked by programmers, or run-time interactions between objects that are parts of the system being developed.

2. Permanent relationships between a pair of programmed objects, such as inheritance or client-supplier relationship as reflected by their scripts.

In the abstract LGA model, inter-object interactions are presented as a *message passing* abstraction, which is to say that objects interact by sending messages. The correspondence between the message passing abstraction and the above kinds of interactions is as follows. For the dynamic operations initiated by one object on another, the correspondence is trivial, the semantics of such a message can be thought of as a remote procedure call: the sender suspends until the call returns. The correspondence between the permanent relationships between two programmed objects and the message passing abstraction is not so trivial. Such interactions are best thought of as messages between the two objects that are all *sent*, in no particular order, at the moment when they are assembled into an executable system.

Syntactically, messages in Darwin have the form of a Prolog-like [9] term. Variables, denoted by capitalized symbols can be used in a message as a place-holder for returning results to the sender.

### 2.1.3 LGA Primitive Operations

LGA specifies a fixed set of *primitive operations* for manipulating the objects in the object-base, that must be provided by the underlying environment. In Darwin for example, the primitive `$do(create(o2)@o1)` is provided for creating a new object `o2` using `o1` as prototype and the primitive `$do(destroy@o)` for destroying the object `o`. Similarly, the primitive `$do(set(p)@o)` adds the property `p` to the state of `o` and the primitive `$do(recant(p)@o)` removes the property `p` from the state of `o`. The

primitive `$do(install(f)@o)` installs the code in file `f` as the script of `o` and the primitive `$do(deliver(m)@o)` delivers the term `m` as a message to the agent of `o`. If `o` is a programmed agent then this might invoke a procedure in the script. The primitive operations can only be mandated by the law, users cannot directly invoke them.

### 2.1.4 The Law

The law prescribes what should be the actual impact of an interaction (a message in the abstract LGA model) that occurs in a project under its jurisdiction. In response to an interaction $i$, the enforcement mechanism $\mathcal{E}$ computes a *ruling*, which is a (possibly empty) sequence of primitive operations, that is then executed. The ruling may depend on $i$ itself, the law $\mathcal{L}$ and on the global control state of the project. If $\mathcal{I}$ is the set of all possible interactions, $\mathcal{CS}$ is the set of all possible global control states, and $\mathcal{R}$ is the set of all possible sequences of primitive operations, then the law $\mathcal{L}$ can be viewed as:

$$\mathcal{L}\colon \mathcal{I} \times \mathcal{CS} \longrightarrow \mathcal{R}.$$

Of course, in any practical implementation, the law cannot be represented purely by extension, that is, by listing the ruling explicitly for each possible interaction and state combination. In general, the law is specified by intention by means of a collection of rules.

In Darwin, a rule is like a Prolog clause. Consider, for example, the following Darwin rule:

```
sent(S,set(X),T):- owner(S)@T, $do(set(X)@T).
```

Note the presence of the term `owner(S)@T` and the Darwin primitive `$do(set(X)@T)` in the rule body. Terms of the form `p@o` succeeds if the property `p` is present in the state of object `o` at the time of ruling computation.

Without getting into details, the above rule governs messages of the form `set(X)` sent by a Darwin object to another, where `X` is a Prolog variable that can unify with any Prolog term. The ruling for a message `set(foo)` sent by `s` to `t` will be

`$do(set(foo)@t)`, only if `s` is the owner of `t` (denoted in `t` by the property `owner(s)`). The result, in that case, will be the addition of the term `foo` in the exterior of `t`.

### 2.1.5 The Enforcement Mechanism

The job of the enforcement mechanism is to compute the ruling for an interaction and then make sure that the primitive operations mandated by the ruling are carried out. There are two basic modes for law enforcement: *by interception* and *by compilation*. They are complementary in a number of ways, and a practical enforcement mechanism is likely to combine both modes. The messages sent by an un-programmed agent or messages sent at run time by the programmed agents can be enforced by interception, which means the messages are intercepted by the enforcement mechanism in order to compute and execute the ruling. The messages that represent permanent relations between programmed objects as defined by their scripts, on the other hand, are best enforced by compilation. We will elaborate on the enforcement aspects in later chapters.

### 2.1.6 Legislation

Every software development project starts under LGA with the definition of its *initial law*, which defines the general framework within which the project is to operate and evolve. Analogous to the constitution of a country, the initial law establishes the manner in which the law itself can be refined and changed during the lifetime of the project. The process of changing the law by means of addition or deletion of rules is known as legislation in LGA.

Legislation is a very critical aspect of LGA and details of the legislative mechanism of a law-governed environment depends on the particular implementation. We will present the legislation mechanism of Darwin-E, which borrows heavily from the legislation of Darwin, later in this thesis.

### 2.1.7 Language Interface

The language interface is responsible for:

1. deciding how programming language modules are represented by LGA (programmed objects) objects, and

2. mapping programming language constructs found in the scripts into inter-object interactions (conforming to the messages abstraction) in such a way that they can be governed by law.

This makes it possible to have scripts written in different programming languages, provided that the environment is equipped with an appropriate language interface. The language interface may map programming language constructs into messages that will be generated at run-time, or alternatively, it may map them into permanent relationships between programmed-objects. The former necessitates the presence of the law enforcer when the software is run, whereas in the later case it could be run off-line.

The language interface for Prolog is an example of the first approach. It introduces a notion of object-based modularity to Prolog, the script (Prolog code) of a module-object is given its own name space, so that its scope is encapsulated within the module-object itself and they interact by sending synchronous messages among themselves. The language interface for Eiffel, developed as part of this research and employed in the Darwin-E environment, takes the other approach. We will see more about this interface later in this dissertation.

## 2.2 Related Law-Governed Research

The objective of this section is to put the current thesis in proper perspective with respect to other Law-Governed work. The initial ideas of *law* and *Law-Governed Systems* were formed during the work in relation to large Prolog programs [8]. This lead to the development of the initial version of the Darwin system [46, 47]. The concept of Law-Governed Systems were consolidated in [41, 50]. The law-governed approach was applied to formulate different object-oriented structures [51, 52, 61, 62].

The law-governed approach has been applied to other problems, such as imposing protocols on distributed systems [42], or ensuring integrity by adding obligations to privileges [39]. The notion of LGA, as a software architecture, was formalized in [43].

In the meantime, our work with software engineering of object-oriented systems began. Compile-time enforcement of the law was developed [57] and the concept of regularities emerged [44, 45, 49].

In this research, we used LGA as the underlying architecture for imposing regularities. This is different from earlier law-governed work related to object-orientation because here we focus on software engineering constraints in large object-oriented projects as opposed to law-governed implementation of isolated objected-oriented concepts.

# Chapter 3

# The Darwin-E Environment: An Overview

Darwin-E is a prototype environment for developing Eiffel software under LGA. In this chapter we will present an overview of the environment, deferring the details for later chapters. We begin with a discussion on how Darwin-E implements a specialization of the abstract LGA model for Eiffel.

## 3.1 Darwin-E As a Specialization of the Abstract LGA Model

First, the objects in the abstract LGA model are refined in Darwin-E to represent various entities relevant to software development in Eiffel. We will describe the Darwin-E objects in Chapter 4.

Second, Darwin-E provides a set of commands for performing various operations such as creation or destruction of objects, assembly and running of Eiffel systems, etc. These commands completely encapsulate LGA primitives [1], external tools like the Eiffel compiler, emacs etc. and the various services provided by the Darwin-E environment such as configuration management.

Third, the message-passing abstraction of inter-object interactions in the abstract LGA model is specialized to present only the following two types of interactions described below:

1. Darwin-E commands issued by un-programmed objects that represent human participants in a project, and

[1] LGA primitives, in general, cannot be used directly in Darwin-E, although certain advanced and highly specialized LGA primitives can appear in user-defined rules, refer to Section 5.3.3 and Section 7.6 for more details.

2. A number of well-defined interactions between Eiffel classes including static inter-class relations such as parent-child (inheritance) or client-supplier, as well as dynamic inter-class relations that reflect some run-time activities in terms of instances of such classes (such as call or assignment),

## 3.2 Structure and Organization of Darwin-E

The root of the Darwin-E environment consists of a directory service named `super-node`. It runs as a *daemon* process and keeps track of the various projects, active or stored, managed by the environment. Each project is managed by a dedicated *project server*, which is allocated by the directory service. Multiple developers can work concurrently on each such project from different physical locations over a local network by connecting to the project server as *clients*. The state of the project is maintained in an *object-base*, managed by the project server. The project server is also responsible for enforcing the law of the project.

The server and clients run over a network of sparcstations communicating via TCP/IP. The client shells use the X window system for various diagnostic messages from the project server as well as for different X-based Eiffel tools. Figure 3.1 illustrates the structure of the Darwin-E environment. It shows two active projects `project 1` and `project 2`, whose servers run on different machines. The directory service runs on another machine. Users working on projects are on different machines on the network[2].

## 3.3 Getting Started With Darwin-E

A user wishing to connect to a project $\mathcal{P}$ first starts up a special shell that communicates with the directory service (refer to the dotted lines in Figure 3.1, denoting initial communication between the shells and the directory service).

If $\mathcal{P}$ is not known to the directory service then it must be a new project, and a new project server is created on a machine known *a-priori* to the directory service, and

[2]Details about these and various other implementational aspects of Darwin-E can be found in Appendix C.

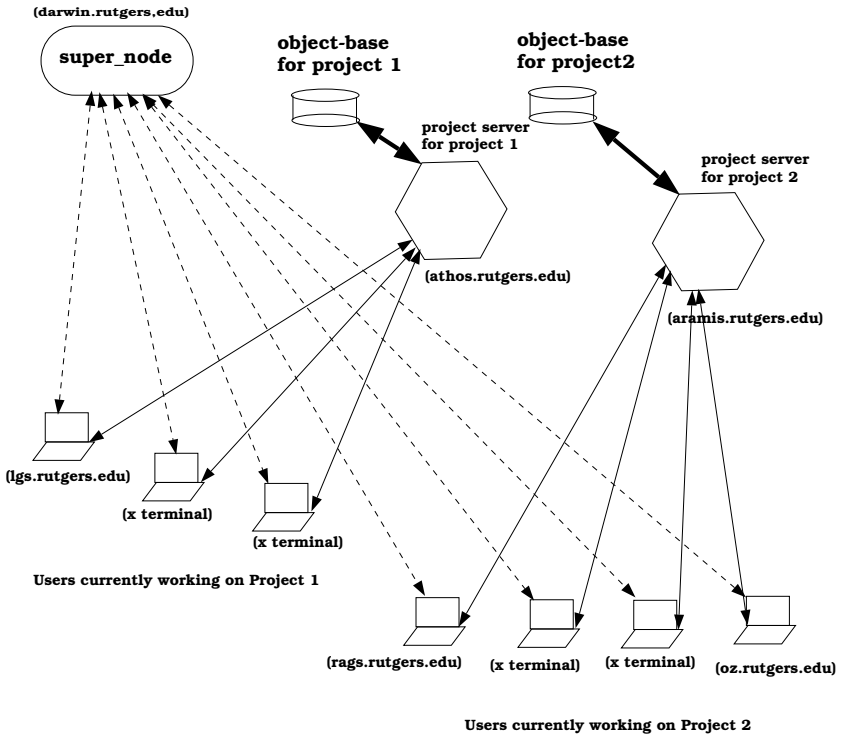


Figure 3.1: Structure of The Darwin-E Environment

the user is connected to that server as the *initiator* of the project. If $\mathcal{P}$ was known to the directory service and there was a project server already running somewhere in the network, then the user gets connected to that project as *newcomer*. If $\mathcal{P}$ was known, but there was no server active for it, then a project server is started by retrieving the stored state of $\mathcal{P}$, and the user is connected to it as *newcomer*.

The *initiator* of a project is responsible for initializing the new project. Initialization of a project includes creating project-specific rules and other objects, and the *initiator*, by default, has the power to do so. The *newcomer*, on the other hand, is not allowed to take part in the project unless he is authenticated by the law of the project.

## 3.4 Software Development and Imposition of Regularities Under Darwin-E

Simply stated, software development under Darwin-E consists of representing Eiffel classes that are developed off-line – outside Darwin-E, in the object-base and putting

them together in a system. These are achieved by means of the Darwin-E commands. This simplistic view assumes that the phases in software life-cycle that precede system assembly, such as design and analysis, are done off-line, outside Darwin-E. Darwin-E therefore does not provide for complete process modeling or enactment like traditional process centered environments. However, the law-enforcement mechanism of Darwin-E is customized to provide a flexible means to control builder activities. In this sense, Darwin-E provides for controlling certain aspects of the software process. We will see more about software development and its control in Chapter 5.

Imposition of regularity under Darwin-E relies on the following two facts: a) regularities are expressed as rules in the law, and b) Eiffel systems constructed under Darwin-E, must satisfy such rules in order to be assembled (and run). How this is done is described in details in Chapter 6.

## 3.5 Law Enforcement and Legislation in Darwin-E

Broadly speaking, the law of a project under Darwin-E consists of two distinct parts:

1. The *evolution sub-law*, which governs the process of development and evolution of the system, and the law itself. Interactions initiated by human users (we will often refer to these as *messages*) are under its jurisdiction.

2. The *system sub-law*, which governs the structure and behavior of the system under development. Interactions between objects that represent Eiffel classes (we will often call these *eiffel interactions* or simply, *interactions*) are under its jurisdiction. Rules that express regularities actually belong to this sub-law.

Although these sub-laws are structurally similar, they are enforced very differently, as illustrated in Figure 3.2. The system sub-law is enforced mostly statically, when the individual program-modules are introduced or changed, or when a system is assembled into a runnable program. The evolution sub-law, on the other hand, is enforced, dynamically, by interception, when a command is issued. More details on this can be found in Chapters 5 and 6.

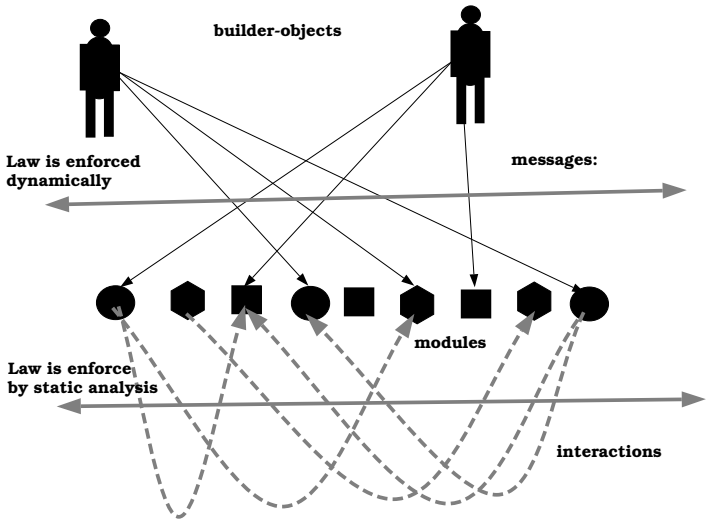


Figure 3.2: Messages and Interactions in Darwin-E

The fact that the law governs its own evolution, involves several subtleties. This pertains to what is known as *legislation*. The issues involved in legislation are quite complex and are the topic of an independent chapter (Chapter 7). We will postpone any discussion of legislation until then.

## 3.6 Other Features of Darwin-E

Configuration management is an important aspect of any large software project. It involves how different versions of modules (in our case Eiffel classes) are managed in the project database and are used to put together in a deliverable software system. We will discuss configuration binding and version management aspects of Darwin-E in Chapter 8.

Any large and long lived software often needs *monitoring*. By monitoring, in this context, we mean one or both of the following two very different matters: (a) monitoring various activities of the personnel involved in development and maintenance of the software, and (b) on-line monitoring of the software itself. In Darwin-E, we incorporated facilities for both, as we shall see in Chapter 9.

# Chapter 4

# The Darwin-E Objects

In this Chapter, we describe the various Darwin-E objects that the object-base $\mathcal{B}$ of a project under Darwin-E may contain. Some of the objects in $\mathcal{B}$ are built-in, i.e. mandated by the environment, some others are created when a project is set up and the rest are created dynamically during the lifetime of the project.

Among the built-in objects, are the three Darwin-E classes `builder`, `module` and `configuration`, which are used as templates for creating new Darwin-E objects. In addition, there are two special purpose built-in objects named `helper` and `#tracer`. The `helper` provides answers to various queries such as which configurations include a given class, which module-objects are there in the object-base, which user-defined rules are active in the project, etc. The `#tracer` is used for storing information about various user activities that are being traced (monitored) in the project. We will see more about this later in Chapter 9. There are a few other built-in objects, that we will mention in due course.

Each Darwin-E object is essentially a refinement of the abstract LGA object. Therefore, they can have various properties (attributes). Some of the properties of Darwin-E objects are built-in, that is, they are mandated by Darwin-E itself, and have predefined semantics; others are mandated by the law of a given project, which defines their semantics for this particular project. Manipulation of the built-in properties can only be performed by Darwin-E itself, whereas, the law of the project at hand governs the manipulation of the user-defined properties.

As an illustration of built-in and user-defined properties, consider an object created from the built-in Darwin-E class `module`. As we shall see, such objects are known as

module-objects, and are used in Darwin-E to represent Eiffel classes. Each module-object `m` has a built-in attribute `className(C)`, where the value `c` of the variable `C` is the name[1] of the class represented by the object `m`.

To illustrate the nature of properties that may be mandated by the law of a given project, let us now introduce a property which will be used later. A property `cluster(x)` of a module-object `m` can be used to indicate that the object `m` belongs to a cluster called ``x''. A distinction between module-objects based on this property can be made from the law as follows: a class in cluster `x` cannot inherit from a class that is not in the same cluster. Note that the same property can be used very differently in the law of another project, which for instance may prevent instantiation of classes in cluster `x`.

We will present more examples of both kinds of properties and their use in the law, as we go on. Let us now present the different kinds of Darwin-E objects that the users can create.

## 4.1 Builder-objects

Builder-objects, representing human personnel involved in the project are instances of the built-in Darwin-E class `builder`. Builder-objects are normally inactive, unless a client of the project-server is associated with it. Darwin-E employs a password[2] scheme to control the association of a builder with a client shell. The *initiator* of a (new) project is always associated with the built-in builder `#creator`, who is empowered to create rules and other objects to formulate the initial state of a project. The shell of a *newcomer* to a project is always associated with the built-in builder `#newcomer`. This `#newcomer` has essentially no power to do anything other than attempting to associate the shell with other builders. Therefore, upon opening the client-shell for an existing

[1] In general, the object-base may have several objects with the same class name, which may represent several versions of the same class. But for simplicity unless specified otherwise explicitly (for example in the context of version management), we shall assume that all class names are unique, and identical to the identifier of the objects representing them.

[2] Each builder is assigned its own name as its default password upon creation. A builder can change its default password by sending appropriate messages (described in the appendix) to itself.

project $\mathcal{P}$, the first activity of a typical user is to associate himself with a valid builder of the project $\mathcal{P}$.

## 4.2 Module-objects

Module-objects, representing eiffel classes are instances of the built-in Darwin-E class `module`. A module-object `m` can be in either of the two states: 1) *attached*: the code defining the Eiffel class represented by `m` is attached to it and 2) *un-attached*: no code is attached to `m`, however it still does represent some class and any Eiffel code defining that class can potentially be associated with it.

Module-objects are created in *un-attached* state. The code (developed off-line) is associated with it by a later action. The name of the Eiffel class a module-object represents, is specified when the module-object is created and cannot be changed during its life. Several module-objects representing different versions of a same class may co-exist in the object-base.

It is possible to attach code to a module-object (we will often refer to a module-object as a class-object or a class) that is already in attached state. As a result of such *re-attachment*, the old code is discarded. This might make the module-object obsolete in the systems that had included it; the ramification of this is discussed in Chapter 6.

Module-objects usually have many built-in and user-defined (builder-defined) attributes. The attributes associated with an example module-object along with the code attached to it is presented in Figure 4.1. The built-in (or reserved) attributes of module-objects are sub-divided into two sets: *intrinsic* and *interface-related*. Among the *intrinsic* attributes are the (primitive) attribute storing the unique identifier of the module-object and the attributes storing the name of the class it represents, its creator, various associated files, etc. The path `p` to the file containing the code associated with a class-object in attached state is also stored as an intrinsic attribute.

The *interface-related* attributes store the interface and structure of the eiffel class that the module-object represents in some disseminated form. The information is stored in terms of class names. For example, if an eiffel class `c` inherits from another class

Definition of a class MYCLASS stored in ~partha/eiffel/myclass.e

```
class MYCLASS
    inherit MYPARENTCLASS
  feature
      a,b: REAL;;
      test0(X:INTEGER) is do
            ...
            ...
      end;
      test(x: INTEGER) is do
            ...
            test0(x);
            ...
      end;
end -- class MYCLASS
```

Attributes of a class-object **o** created by builder **jack** are shown below in **bold face** with brief comments after --

Intrinsic Attributes:

```
#id(o)      -- denotes the id of this class-object
className(myclass) -- denotes the name of the class it represents
eiffelSource('~partha/eiffel/myclass.e') -- path to the body
creator(jack) -- id of the builder who created it
```

Interface-related Attributes:

```
inherits(myparentclass)
defines(attribute(a),type(real))
defines(attribute(b),type(real))
defines(x,of(integer),as_arguments_of(test0))
defines(routine(test0),type(no_type))
defines(x,of(integer),as_arguments_of(test))
defines(routine(test),type(no_type))
exports(feature(a),[all])
exports(feature(b),[all])
exports(feature(test0),[all])
exports(feature(test),[all])
```

Builder-defined Attributes:

```
owner(john) -- represents the builder who owns the class-object
cluster(application) -- denotes the cluster to which it belongs
tested   -- denotes that this class has been tested satisfactorily
```

Note the use of lower-case letters in the attributes, since the attributes are prolog terms, we cannot use upper-case letters which would denote a prolog variable.

Figure 4.1: Example of Attributes in a module-object in attached state

`d`, then class name `d` is used in the attribute `inherits(d)` of class object `o` representing class `c`. Module-objects in *un-attached* state does not have the *interface-related* attributes. Note that it is possible to attach incomplete code to a module, such as a partially defined Eiffel class. In such cases Darwin-E tries to extract as much interface-related attributes as it can during the attachment process.

The builder-defined attributes represent various project-specific information associated with the module-objects that are not related to Eiffel. For instance, the name of the owner of the module-object or the cluster to which it belongs, are stored as attributes in the above example. As we shall see, these attributes will often be used in our regularities.

## 4.3 Configuration-objects

A 'system' in Eiffel terminology refers to an executable program resulting from a collection of classes put together. A single project usually gives rise to many different 'systems' at different times. In Darwin-E, a configuration-object, which is an instance of the built-in Darwin-E class `configuration`, represents an Eiffel system. A configuration-object is a specialization of the *idle object* in the abstract LGA model. It does not have any script associated with it and serves as the repository of various information (that are stored as attributes) about the system it represents. A configuration-object `c` is associated with a directory. The built-in attribute `directory(p)` (where `p` is the complete path to the directory) in `c` represents this association. A configuration-object *includes* the module-objects (representing Eiffel classes) that constitute the Eiffel system it represents. A distinction is made however, between user-defined classes and library classes. Module-objects representing user-defined classes *must* be explicitly included in the configuration. This can be done either by the builder directly putting in module-objects in the configuration or by the configuration binding mechanism (described later in Chapter 8). On the other hand, users need not create and include module-objects that represent Eiffel Library classes; Darwin-E ensures that appropriate representation of the library classes are found in every configuration.

If a module-object `m` representing a user-defined class `mclass` is included in a configuration `c`, Darwin-E copies the Eiffel code associated with `m` to a file `mclass.e` in `c`'s directory[3]. Two module-objects representing a same class cannot be *included* in a configuration. The directory associated with a configuration essentially serves as the home of the root-cluster of the system represented by the configuration. The customization of the `Ace` file required for this is automatically performed by Darwin-E.

A module-object `m` included in a configuration can become obsolete as a result of code re-attachment, because `m` is associated with new code now, whereas the configuration directory still contains the old code for the class represented by `m`. Darwin-E automatically gets the fresh code when the configuration that contains the obsolete module-object is enforced or compiled again.

## 4.4 Group-Objects

Usually there are multiple incarnations of a single class in an object-oriented project. A group-object stands for a set of module-objects that are different incarnations of the same class. While building a configuration, groups may be specified instead of actual classes, leaving the selection (configuration binding) of module-objects for later. Group-objects also do not have any script; they are merely a tool to organize module-objects with the same `className` attribute.

A group-object for modules representing class `c` does not exist by default it has to be created explicitly as an instance of the built-in Darwin-E class `group`. The name of the class whose incarnations are collectively represented by a group-object is associated with it at the creation time. Even if a group-object `g` for different incarnations of class `c` is created, every module-object representing class `c` does not by default belong to `g`. One has to `enroll` module-objects into the corresponding group. For example, module-objects `o1,o2,o3` and `o4` may all represent (different incarnations of) class `c` in the object-base of a project, and yet only `o1` and `o2` may belong to the group `g`.

Enrolling a module-object in its group implies that the particular incarnation of

[3]That the file-name storing eiffel code for class `x` must be `x.e` is a requirement of the Eiffel compiler.

the class it represents is submitted to the project-wide repository. Such a class may be chosen during configuration binding, even when the configuration builder may not know about its existence. A module-object not enrolled in its group on the other hand, represents an incarnation of a class in isolation. In order to use it in a configuration, the configuration builder must include it by name. Attempts to enroll a module-object to a group that is not meant for the same class result in an error. A module-object can be `withdrawn` from its group, if desired. When a module-object `o` is enrolled in a group `g` it is noted as the latest object to be in the group by an attribute `current(o)` in `g`.

The notion of groups provides an open framework for law-governed version management. By controlling the enrollment of module-objects into groups we can devise sophisticated version control schemes that can work coherently with our consensus based configuration binding scheme. This topic is discussed separately in Chapter 8.

## 4.5 Rules

The law consists of individual rules that are Prolog like clauses. Every rule-object encapsulates one of the clauses of the law of the project. This makes it easier to understand and control the very important process of legislation, discussed separately in Section 7. We will see more about rule-objects there.

## 4.6 Metarules

Metarules are the objects that play an important role in the creation of rules. They are also discussed in detail in Section 7.

# Chapter 5

# Software Development in Darwin-E

The term software development is used in a narrow sense in the context of Darwin-E: it refers to the sequence of activities performed by the builders involved in the project in its object-base. Such activities include creating new objects, attaching or editing code to module-objects, setting or modifying properties of objects, compiling or running Eiffel systems and so on. Each such activity is associated with a Darwin-E *command*, and as we mentioned earlier, Darwin-E provides a fixed set of such commands.

A builder `b` invokes the command `m` on a target object `t` by sending the *message*[1]

```
b:  ^m => t.
```

Messages can be sent by the builders either individually or by invoking a script which sends a series of messages in a predefined sequence. Each such message is intercepted by the law-enforcer $\mathcal{E}$ at the project-server as a `sent(s,m,t)` goal, which is evaluated against the law (the evolution sub-law to be precise; in this chapter, by *law* we normally mean this subset of the law of the project, unless specified otherwise) to compute the *ruling* for this message. The ruling may either reject the message or accept it. The activity associated with the command involved in the message is executed only if the ruling accepts the message. This makes it possible to specify by law which commands will be executed at all in Darwin-E. Furthermore, an accepting ruling may prescribe certain additional operations which are also carried out along with the execution of the command issued as the accepted message. A rejected message simply produces an error notification for the builder who issued the command.

---

[1] The use of ^ as prefix to the command in the message is purely for syntactic and implementational reasons.

## 5.1 Darwin-E Commands

The repertoire of Darwin-E commands are best described by dividing them into several categories[2]. One such category consists of a set of special commands, used for connecting and disconnecting with a project-server and associating a builder-object with a human user. We will call these the **Connection Commands**. These commands are different from the commands in other categories in the sense that they are not subject to the law of the project. We will say more about these commands in Section 5.1.1 after we describe the other command categories.

- **Creation Commands:** These are used for creating module-objects, builders or configurations. In order to create such an object, a builder sends a `new(..)` command to the appropriate Darwin-E class. The number and the nature of the arguments of this message vary for the different targets.

- **Observer Commands:** These commands are used by the builders for observing properties of objects or for reading code attached to module-objects.

- **Modifier Commands:** Commands in this category are used to change the state of the object-base. The kinds of changes we consider here include adding or removing (user-defined) properties to or from an existing object, making an object persistent or unpersistent, attaching or changing the code to a module-object ( this may result in addition or removal of several built-in properties) or removing non-rule objects (removing rules and metarules are considered under legislative commands)[3].

- **Eiffel Commands:** The ISE eiffel 3.1 comes with an environment that provides various tools that could be invoked from the Unix operating system. This environment is embedded in Darwin-E. In `Darwin-E` such tools are invoked by a set of commands that comprise this category.

---

[2] A glossary of Darwin-E commands can be found in appendix C.

[3] Of-course, the state changes also when a new object is created but we treat creation of objects separately.

- **Configuration Commands :** Commands in this category effect some kind of operation with respect to the configuration. The most important operations regarding a configuration `c` are *putting classes into* `c`, *enforcing the law of the project on* `c`, *compiling* `c` to create an executable and running it. Note that operations performed by some of the configuration commands may change the state of the object-base. For example, *putting classes* into a configuration changes the `systemModel(_)` and `baseLine(_)` properties of the configuration, *enforcing a configuration* `c` implies the addition of the property `enforced` to `c`, and so on. These occur as side effects of configuration specific activities and the properties affected are (built-in) properties reserved by Darwin-E.

- **Trace Related Commands:** Commands in this category are used to view and remove various trace information stored by the built-in utility `#tracer`. We will see more about these in Chapter 9.

- **Legislative Commands:**This category contains commands for creating and destroying rules and metarules. The operation of creating rules/metarules is a critical one and is quite involved. We will discuss it in Chapter 7.

Let us now present some examples. In these examples, we will show how to issue the appropriate commands as messages in order to perform specific operations, and describe the effects assuming the law returns accepting rulings.

### 5.1.1 Example Of Using Commands

A user needs to register herself with the Darwin-E environment (which runs as a daemon process always) by typing in the connection command `wcli` at the Unix prompt. This starts up the shell which will be acting as a client of the project-server. To connect to a project `p`, she will then type in another connection command `activate(p)` at the shell just started. If Darwin-E has started a project-server for the project `p`, then the user's shell will be connected to this project-server as the built-in builder `#newcomer`. She can then identify herself with the builder designated for her by using another connection command `^enter`.

Once connected to a project, a typical programmer Jill may create a builder to represent another programmer Phill as follows:

```
jill: ^new(phill) => builder.
```

She may create a module-object `o` to represent the eiffel class `point` as follows:

```
jill: ^new(o,point) => module.
```

To attach file `f` to the module-object `o` she would do:

```
jill: ^attach(f) => o.
```

This results in setting the interface-related properties of `o`, which is obtained by a fast scan over the code stored in `f`. She can look at these and any other properties that `o` may have by issuing:

```
jill: ^print => o.
```

Similarly,

```
jill: ^more => o.
```

will print the code attached to module object `o` on her screen. To edit the file later, she would do:

```
jill: ^emacs => o.
```

This will launch the emacs editor with the file attached to `o` as a separate process so that the server can process other messages.

To create a group-object `g` to stand for all incarnations of the class `point`, she would send the following message:

```
jill: ^new(g,point) => group.
```

To include the module-object `o` in the above group, she would do:

```
jill: ^enroll(o) => g.
```

To create a configuration `cfg` with the directory `d` as its workspace, she would send the following message:

```
jill: ^new(cfg,d) => configuration.
```

To put several module-objects into the configuration `cfg` she would do:

```
jill: ^put([o1,o2,o3,o4,o5,o6,o7]) => cfg.
```

The above explicitly includes the module-objects listed as the argument of the `put` in `cfg`. As an alternative, she can use groups for those classes she is not sure which particular incarnation to include:

```
jill: ^put([(pixel,_),(line,_),o3,o4,o5,o6,(figure,_)]) => cfg.
```

In this example, `pixel`, `line` and `figure` stand for the groups that represent different incarnations of the `pixel`,`line` and `figure` classes[4]. Appropriate module-objects are to be chosen later for each such group during the configuration binding process which can be initiated by the following message:

```
jill: ^bind => cfg.
```

There are many salient features in this process which are discussed in details in Chapter 8. Note that binding is not required if module-objects are directly put into the configuration using the `put` command.

At this point, the configuration is ready to be compiled and run. But, Darwin-E does not allow a configuration to compile or run unless the law is enforced upon the Eiffel interactions it contains. To disallow others to modify and use a configuration when it is being enforced, Darwin-E requires the builders to lock the configuration before enforcement. To lock `cfg`, `jill` would send the following message:

---

[4]It is not necessary that the group should be named after the class it represents, however, we will use this convention for simplicity.

```
jill: ^lock => cfg.
```

This will lock the modules representing the classes that are explicitly included in it, provided they are not already locked by anybody. Then, `jill` can proceed to impose the law of the project on `cfg`:

```
jill: ^enforce => cfg.
```

This step consists of static analysis of Eiffel code and consultation with the law, and may take considerable time to complete. If it is carried out at the server, all the builder activities will suspend until this completes. As a remedy, this step is carried out as a separate process at `jill`'s machine, so that the project-server is free to respond to messages from other builders. After this process terminates, if she wants to *freeze* her configuration (one compiler option provided by ISE eiffel 3.1) she would do:

```
jill: ^freeze => cfg.
```

This will run the Eiffel compiler at `jill`'s machine. Finally, after the compiler terminates successfully, she may run the executable thus created:

```
jill: ^run => cfg.
```

This will create a new window at `jill`'s machine in which the executable will run.

This concludes our description of Darwin-E commands. We will now turn our attention to several technical aspects of the client-server mechanism of Darwin-E.

## 5.2 Technical Aspects of the Client-Server Mechanism

### 5.2.1 Need For Transaction Semantics

Some of the commands involve simple atomic operation. For example, consider the modifier command `set(p)`, the property `p` is either set or not, there cannot be any intermediate state. But, there are other commands that may involve a sequence of such atomic operations. For example, consider putting a list of classes into a configuration:

not only does it involve copying the associated files into the configuration directory, but also setting or changing various properties of the configuration at hand. It is possible that any of these atomic operations will fail, because of unpredictable and unforeseen hardware or software resource problem. This means that, in the presence of failure, such a command may have different partial executions, which can lead to potential inconsistency. Ideally, in such a situation we would need a *all or nothing* transaction semantics: if any operation in the sequence fails, the command should *roll-back* and the state of the system should be recovered to the state before the command was initiated.

We did not implement the *all or nothing* semantics in the prototype for a number of reasons. First, the issue (all or nothing semantics of commands) is orthogonal to the basic premise (law-governed regularity) of this research. Second, the problem involved and the solution is well understood and it is only a matter of implementation, which is unlikely to benefit this research. Third, even without the roll-back and recovery mechanism, the prototype is able to achieve its objective in demonstrating the use and benefits of law-governed regularity in object-oriented software projects. We provide diagnostic messages to report the failure of component atomic operations associated with a command and rely on the users to manage the recovery. For example, a configuration may have been created but its associated directory could not be created because of access permission. In this case the builder trying to create the configuration will be informed about the problem and then she may destroy the configuration just created and try to create it with a different path.

### 5.2.2 Need For Launching Separate Processes

The server handles the messages one at a time, the processing of another message cannot start unless the server is done with the (complete or partial) execution of the command encapsulated in the current message. If the atomic operations are short then there is no problem, but if the operations are large, for example invoking emacs, then it means that the server is blocked until emacs quits. For this kinds of operations the server launches separate processes (for instance launch emacs in the background) so that it

can proceed onto the next operation. Most of the operations that are handled this way (emacs, compiling a configuration, running a configuration, running Eiffel tools) involve potentially indeterminate delay because of human intervention, and can proceed to completion independently without any communication to and from the server. However, there is one command, discussed next, that takes considerable time to execute and needs close interaction with the server.

Enforcing the law on a configuration is inherently a complex and time-consuming process. It uses and affects the object base a lot, although in a highly focussed and localized manner: it may create new objects but will modify only the objects that belong to the configuration being enforced. The way Darwin-E handles the `enforce` command is as follows. If the message containing the *enforce* command from a builder is approved by law, a new process is created at the machine on which the builder's shell runs. This process is given a copy of the parts of the object-base it needs for doing the enforcement. When the enforcement process completes successfully, the cached objects (as well as the new objects (if any) that are created during enforcement) are merged with the object-base maintained by the server. On the other hand, if this process terminates abnormally (as the result of a failure), no such merging is necessary.

### 5.2.3 Need For Locks

The existence of commands resulting in lengthy activities carried out as separate processes incurs the following problem: objects used by such a command `c` may be seen or modified by other commands while `c` is being executed, leading to potential inconsistencies in the object-base. A locking mechanism is thus required to prevent this from happening. For instance, when a configuration is being enforced, we may not allow any activity on the classes that belong to this configuration.

In Darwin-E, we have implemented a simple locking mechanism that we will describe now. Darwin-E provides the `lock` and `unlock` commands that can be used respectively for obtaining and releasing locks on objects. A builder `b` obtains a lock on an object `o` if the law allows `b` to send a `^lock` $\Rightarrow$ `o` message and `o` is not currently locked by

anyone else. Similarly, `b` must hold the lock on `o` and should be allowed by law to send a `^unlock ⇒ o` message, in order to free `o`.

The above implies that an object can be locked by at most one builder at any time. Locking a composite object, such as a group or a configuration, that is composed of other objects, means that all its components are locked by the same builder. Consequently such an object can be locked only if all its components can be locked. Except for enforcing the law on a configuration (in which case the builder is required to have a lock on the configuration under consideration), it is not mandatory to acquire a lock on an object before using it.

The semantics of locks in Darwin-E can be described as follows. The Modifier Commands, Legislative Commands and the Configuration Commands are sensitive to locks. If a builder `b` issues such a command on an object `o` and `o` is locked, the command will be executed only if the lock holder is `b` himself, irrespective of whether the law approves `b` to send such a message or not. The Eiffel Commands, Creation Commands and Observer Commands on the other hand are insensitive to locks: the server does not care if the builder who issued the command is different from the one who holds the lock on the target. If the law approves the corresponding message, such commands are always executed. The choice of lock-sensitive commands are not arbitrary: these are the commands that can cause inconsistencies if they are allowed to access objects already being used by other commands.

### 5.3 Controlling Software Development Activity

Recall that the activity associated with a command `m` is executed only if the ruling of the law accepts the corresponding message. The message

```
s:  ^m ⇒ t,
```

intercepted as the prolog goal `sent(s,m,t)` for ruling computation, invokes another prolog goal `canDo(s,m,t)`; and is accepted by the ruling only if evaluation of `canDo(s,m,t)` succeeds. This mode of law enforcement is known as *enforcement by interception*. The

`canDo(s,m,t)` goals are defined by the `canDo` rules that the users can create. Therefore, by putting appropriate `canDo` rules in the law one can specify which software development activities are permitted in the project. In the following, we will present several examples of such `canDo` rules. It is also possible to specify what kinds of `canDo` rules are creatable in a project and by whom, but this aspect will be dealt with in Chapter 7.

Before we begin our examples, a brief note about the complexity associated with this mode (by interception) enforcement is in order. In this mode of enforcement, the law is consulted on a per message basis. If there are $n$ rules in the evolutionary sub-law, then in the worst case, the enforcer may need to consult all of them before deciding whether to accept or reject a message. Therefore, the overhead of ruling computation increases with the number of rules. Typically in a project the number of rules are expected to small (in the order of 50-100 rules).

### 5.3.1 Example `canDo` Rules

In order to permit the project manager to create builders, we need the following `canDo` rule[5] :

$\mathcal{R}2$. `canDo(S,new(_),builder) :- role(manager)@S.`

*The property* `role(manager)` *designates a builder as a manager.*

To allow any member of the kernel team to edit any kernel class, we need the following rule:

$\mathcal{R}3$.
```
canDo(S,emacs,T) :-
        cluster(kernel)@S, cluster(kernel)@T.
```

*The property* `cluster(kernel)` *in a builder or a module denotes that it is part of the kernel cluster.*

The `emacs` command is an example of a modifier command. Other members in this category are commands to set and recant properties on objects, attaching code to a module-object and so on. We could create individual `canDo` rules permitting members

[5]In the example rules we will often see terms such as `p@o` - which as mentioned earlier in Chapter 2, evaluates to true if the property `p` is present in the object `o`.

of the kernel team to invoke each command in this category. On the other hand, using the `modifierOp(M)` goal, which is defined by a built-in rule and succeeds if `M` is really a modifier command, we can achieve the same by a single rule:

$\mathcal{R}4$. `canDo(S,M,T) :-`
`cluster(kernel)@S,cluster(kernel)@T,`
`modiferOp(M).`

*In addition to testing the sender and the target for membership to the kernel cluster, this rule also tests whether the command in question is a Modifier Command, before permitting it.*

Similarly, the goal `traceOp(M)`, defined by another built-in rule, succeeds if `M` is the command for viewing or removing the trace information stored in the built-in utility `#tracer`. This could be used to allow the manager to manipulate trace information:

$\mathcal{R}5$. `canDo(S,M,#tracer) :- role(manager)@S, traceOp(M).`

### 5.3.2 Refinement of canDo Rules Based on Command Categories

The built-in goals like `modifierOp(_)` or `traceOp(_)` are provided as a convenience for the users to structure and refine their `canDo` rules based on the different command categories. There are usually many commands that belong to one category. So one may need sub-categorization within a category in order to implement finer control over builder activities. This refinement can go on and on until we reach individual commands. Instead of a flat set of `canDo` rules, one might use a flexible and structured scheme of control as presented in Figure 5.1. The rules in Figure 5.1 clearly demonstrate how to filter out different commands according to their category and subject them to different rules. This adds to the flexibility of our scheme, because for example, modifier commands now can be controlled separately and independently of configuration commands.

### 5.3.3 Prescribing Additional Operations From canDo Rules

Certain additional operations can also be prescribed by a `canDo` rule, by including Darwin-E (LGA) primitives of the form `$do(_@_)` in the rule. However, Darwin-E

$\mathcal{R}6$.
```
canDo(S,M,T) :- modifierOp(M),
        canModify(S,M,T).
```

*This rule filters out* `modifierOp` *commands and subjects them to the* `canModify` *rules.*

$\mathcal{R}7$.
```
canDo(S,M,T) :- configurationOp(M),
        canConfigure(S,M,T).
```

*This rule filters out* `configurationOp` *commands and subjects them to the* `canConfigure` *rules. There will be similar* `canDo` *rules for each of the command categories, however, in this example we will only consider further refinement of* `modifier` *commands.*

$\mathcal{R}8$.
```
canModify(S,M,T) :- type(eiffel)@T,
        canModifyClass(S,M,T).
```

*This rule filters out* `modifierOp` *commands addressed to class objects and subjects them to* `canModifyClass` *rules.*

$\mathcal{R}9$.
```
canModify(S,M,T) :- type(group)@T,
        canModifyGroup(S,M,T).
```

*This rule filters out* `modifierOp` *commands addressed to group objects and subjects them to* `canModifyGroup` *rules. There will be other such* `canModify` *rules, but henceforth in this example we will consider further refinement of modifier commands addressed to class objects only.*

$\mathcal{R}10$.
```
canModifyClass(S,M,T) :-
        not (M == emacs),
        cluster(X)@S,
        cluster(X)@T.
```

*This rule allows a programmer to use any* `modifierOp` *command on classes belonging to his cluster, except for editing.*

$\mathcal{R}11$.
```
canModifyClass(S,emacs,T) :- owner(S)@T,
        cluster(X)@S,
        cluster(X)@T.
```

*This rule allows only the owners to edit.*

Figure 5.1: Refinement of Categories

allows only a few safe operations to be prescribed this way. As an example of prescribing additional operations while permitting an activity by a `canDo` rule, let us consider the following rule:

$\mathcal{R}$12. `canDo(jill,M, module) :- M= new(_,_),`
`$do(trace(jill,M,module))@ #tracer.`

*This rule demonstrates the use of an LGA (Darwin-E) primitive in an user-defined rule.*

The term `$do(trace(s,m,t))@ #tracer` is a Darwin-E primitive that stores the message `s: ^m` $\Rightarrow$ `t` in the built-in utility `#tracer`. The above rule permits the builder `jill` to create new module objects, and in addition prescribes that such an activity be traced. There are a few other primitives that can be used in a similar fashion with the objective of tracing developer activities; we will discuss them in Chapter 9.

Primitives for manipulating user-defined properties are also permitted in a `canDo` rule. Consider the following rule for example, that allows the kernel programmers to create new classes, but marks them to be kernel classes:

$\mathcal{R}$13. `canDo(S,new(O,_),module) :- cluster(kernel)@S,`
`$do(set(cluster(kernel))@O).`

*Modules created by kernel programmers belong to the kernel cluster, an LGA (Darwin-E) primitive is used in this rule to mark the newly created modules with the user-defined property* `cluster(kernel)`.

Recall that `cluster(kernel)` is a user-defined property. Darwin-E will not permit a `canDo` rule that sets a built-in property such as `inherits(_)`, rather than `cluster(_)`.

As another example consider the following rule:

$\mathcal{R}$14. `canDo(S,new(O,_),group) :- role(manager)@S,`
`$do(set(currentVn(1))@O).`

*Managers are allowed to create group objects, but upon creation each group is made to have the property* `currentVn(1)`.

The property `currentVn(1)` in a group can be used as a seed for generating version numbers (refer to Section 8.1.3 for details) for its members.

# Chapter 6

# Imposing Regularity Under Darwin-E

We would like to be able to specify and impose the desired regularities of a system $\mathcal{S}$ by means of rules in the law (system sub-law to be precise; in this chapter by *law* we will normally refer to this subset of the law) of the project that develops $\mathcal{S}$.

In Darwin-E, such rules regulate a set of interactions between module-objects that represent Eiffel classes, ranging from static relationships (such as inherits ) to compile-time view of runtime activities (such as call). In Section 6.2, we will discuss the Eiffel interactions that can be regulated under Darwin-E and show how various regularities involving them can be formulated as rules.

These interactions are obtained by analyzing the code associated with the module-objects included in a configuration. Consider for example a configuration `c` which includes two module-objects `o1` and `o2` representing classes `c1` and `c2` respectively. If class `c1` inherits from class `c2`, then analysis of the code attached to `o1` would reveal an *inherit* interaction, which we will denote as `inherit(o1,o2)`. Other controllable interactions are extracted in a similar manner. Note that the interaction is defined in terms of module-objects as opposed to class names and the context of the static analysis is a configuration, rather than the object-base. This is so because there may be many module-objects representing a single class in the object-base of the project, and only one of them can be actually included in a given configuration. In many examples, however, we will assume that the identity of a module-object is the same as the name of the class it represents, for reasons of simplicity.

A distinction is made however, between user-defined and library classes in this regard. A library class is not analyzed to extract its interactions with others, which

excludes interactions originating from library classes from the scope of our regularities[1]. The rationale behind this decision is that we assume that the vendor supplied libraries are well written and are not likely to benefit from regularities. Furthermore, even if a library class does not satisfy the law, we are unlikely to fix the problem: the source code may not be available and even if it is, it may be dangerous to modify the library class itself because of dependencies with other library classes.

It is possible, however, to regulate the interactions of user-defined classes with library classes. It may also be possible to regulate interactions of library classes with others, if source-code of the library classes are available: one can always copy library classes into user directories and treat them as user-defined classes.

The enforcement of law upon these interactions works much like a static type-checker: Darwin-E will allow compilation of a configuration only if the interactions between classes included in it satisfies the law. If changes are made to a system which is already enforced, Darwin-E forces a re-enforcement before it can be recompiled. A re-enforcement is not imposed, on the other hand, if the law changes. The technique for enforcing rules about regulated Eiffel interactions is known as *enforcement by compilation*, and was developed in [57]. The following is a brief description of how this is done in Darwin-E.

## 6.1 Compile-Time Enforcement of Rules on Regulated Eiffel Interactions

For each class (module-object) `c` included in a configuration, a set of binary interactions are extracted by static analysis[2]. Let `t(a1,a2,...,an)` denote an arbitrary interaction (such as the `inherit(c1,c2)` interaction introduced above). For each such `t(a1,a2,...,an)`, Darwin-E evaluates the *goal* `t(a1,a2,...,an)` with respect to the system sub-law, to compute the ruling for this interaction. This ruling may have one

[1] Note that this does not prevent us from controlling the use of library classes in user-defined classes.

[2] The static analysis of source code to identify the interactions and the consequent enforcement of the law is a time consuming process. We have incorporated various optimizations for this process which are discussed in Appendix D.

of the following consequences:

1. The interaction may be *rejected.*

2. The interaction may be *admitted.*

3. The interaction may be *admitted with some changes.*

In case of a rejection, the offending module-object `c` is declared *illegal*, and a system with any illegal module are not allowed to compile.

In the present Chapter we will mostly be concerned about the first two effects, and the third effect will be discussed in Chapter 12. Let us now explain how the users can define rules to control regulated Eiffel interactions .

### 6.1.1 Regulating Eiffel Interactions By User Defined Rules

For every interaction `t(a1,a2,...,an)`, there is a built-in rule in the law of every project under Darwin-E of the following form:

```
R. t(A1,A2,...,An):-
     cannot_t(A1,A2,...,An)  ->
      $do(error(['interaction  ', t(A1,...,An),' prohibited'])) |
         (can_t(A1,A2,...,An) -> true |
             $do(error(['interaction ', t(A1,...,An),' not permitted']))
           ).
```

The evaluation of a goal `t(a1,a2,...,an)` with respect to the system sub-law starts with invoking the Rule $\mathcal{R}$ and proceeds as follows: First, the goal `cannot_t(a1,a2,...,an)` is evaluated. If this goal is satisfied, then the interaction `t(a1,a2,...,an)` at hand is rejected. If, on the other hand, the evaluation of `cannot_t(a1,a2,...,an)` fails, then the goal `can_t(a1,a2,...,an)` is evaluated. If this succeeds, then the interaction is admitted. If evaluation of `can_t(a1,a2,...,an)` fails, then also the interaction is rejected.

Roughly speaking, the effect of this built-in rule is that the disposition of an interaction `t(a1,a2,..,an)` is determined by rules of the kind `cannot_t(A1,A2,...,An)` and `can_t(A1,A2,...,An)` (Note the capitalized symbol such as `A1` represent here variables in Prolog sense.). The `cannot_t(A1,A2,..An)` and `can_t(A1,A2,..An)` rules respectively serve as *prohibitions* and *permissions* of interaction `t(A1,A2,..,An)`. These are the rules that the users will define for their projects.

To describe the structure and operation of such built-in rules along with user-defined prohibitions and permissions, let us consider the ruling computation of the interaction `inherit(o1,o2)` introduced earlier. Ruling computation will begin by invoking the $\mathcal{R}$ like built-in rule for inherit interaction. It would then evaluate the goal `cannot_inherit(o1,o2)`. Assuming, for instance, that there exists a prohibition rule like:

$\mathcal{R}$15. `cannot_inherit(o1,_) :- true.`

the goal `cannot_inherit(o1,o2)` would *unify*[3] with the head of this rule, invoking its body. This body would succeed, effecting the rejection of the interaction in question.

The existence of built-in rules of the type defined in Rule $\mathcal{R}$ allows one to choose, for each regulated interaction `t`, one of the following three possible regulation regimes: *prohibitions-based* regime, *permissions-based* regime, and a *mixed* regime.

In a prohibitions-based regime, an Eiffel interaction `t` is permitted unless it is explicitly prohibited. Such a regime is established by defining a default permission for all `t` interactions of the form:

$\mathcal{R}$16. `can_t(A1,A2,...,An) :- true.`

which leaves prohibition rules of the form `cannot_t(A1,A2,...An)` as the only means to regulate `t` interactions.

In a permission-based regime, an interaction is allowed if and only if it is explicitly permitted. This regime is in effect if we do not use prohibition rules at all.

---

[3]Unification is meant here in the Prolog sense. Note again, that a capitalized symbol or an underscore, represents a variable in Prolog, which unifies with any term.

Both prohibitions and permissions are utilized in the mixed regime, in which an interaction `t(a1,a2,...,an)` can be explicitly rejected by a prohibition or implicitly rejected by a lack of permission. The prohibitions take precedence over permission according to the structure of the built-in rule $\mathcal{R}$, which means existence of both permission and prohibition for an interaction `t(a1,a,2,...an)` will result its rejection.

One advantage of prohibition based control over its permission based counter-part is modularity in combining constraints[4]. A prohibition rule, imposing a certain constraint can be freely combined with another prohibition imposing a different constraint, without loosing their individual effects. This cannot be claimed about permissions in general, as explained below.

Consider a constraint $C_1$ that rejects a set $S_1$ of interactions. Now $C_1$ can be imposed either by permission $Pm_1$ that succeeds for interactions in $\overline{S}_1$ (complement of $S_1$) or by a prohibition $Pr_1$ that succeeds for interactions in $S_1$. Similarly, for another constraint $C_2$, rejecting a set of interactions $S_2$, we may either have a permission $Pm_2$ that succeeds for interactions in $\overline{S}_2$ or a prohibition $Pr_2$ that succeeds for interactions in $S_2$. If we want to combine $C_1$ and $C_2$, we would expect that interactions in $S_1 \cup S_2$ will be rejected. If we combine $Pr_1$ and $Pr_2$ this would be the case. Whereas, if we combine $Pm_1$ and $Pm_2$ only interactions in $S_1 \cap S_2$ will be rejected, all other interactions will be permitted by either $Pm_1$ or $Pm_2$. In fact, if $S_1$ and $S_2$ are disjoint, then combining the individual permissions will permit all interactions.

## 6.2 Regulated Eiffel Interactions

In this section we discuss the interactions regulated under Darwin-E. Besides defining each of these interactions, we motivate the need for regulating it, and illustrate such regulation by means of few examples.

Three comments are in order before we start. First, the interactions to be introduced below are not entirely disjoint, in a sense that a given linguistic construct may be viewed

[4]This advantage is partly due to the fact that the constraints we deal with in the context of Eiffel interactions can be naturally modeled as restrictions on certain interactions.

as involving two separate interactions. For example, the Eiffel statement `!!x` is viewed as a `generate` interaction (see Section 6.2.5), because it creates a new object; as well as an `assign` interaction (see Section 6.2.6) because it assigns to `x` a pointer to the new object. Second, we point out that the various subsections below are independent of each other and can be read in any order. Third, for the examples in this section we will limit ourselves to the *prohibition-based* rules. We will assume that the blanket permission rules that are required for this regime to work exist and will not show them. Finally, to make the exposition simple, we will assume that the identity of a module-object is same as the name of the class it represents, which allows us to use `c` interchangeably to refer to the module-object `c` or the Eiffel class `c` it represents.

### 6.2.1 The use of naked C-code by Eiffel Classes

The ability to use code written in the C programming language for the body of a routine of a class is a necessary but very unsafe aspect of Eiffel. Besides providing the ability to make system calls, it can be used to provide various services not provided by Eiffel itself. But routines written in C can also cause the violation of all the basic structures of the Eiffel language, including encapsulation, and thus need to be regulated. For this reason, we define the use of C-code as a regulated interaction in the following manner:

**Definition 1 (useC interaction)** *Given a class* `d` *and a routine* `r` *defined in it, we say that the interaction* `useC(d,r)` *occurs if the body of* `r` *is written in the language C.*

According to the convention introduced above, this interaction is regulated by rules of type `cannot_useC`. For example, the design principle that *only classes in the kernel cluster can use C-coded routines* can be established by including the following rule in the law:

$\mathcal{R}$17. `cannot_useC(D,_) :- not cluster(kernel)@D.`

The effect of this rule is that a class cannot use C-code unless it belongs to the kernel-cluster; or, in other words, that only kernel classes can use C-code.

Another example of control over this interaction is provided by the following rule:

$\mathcal{R}$18. `cannot_useC(D,_) :- owner(P)@D, status(trainee)@P.`

which has the effect that modules owned by trainee programmers cannot use C-code — quite a reasonable managerial restriction.

### 6.2.2 Inheritance

With all its benefits, inheritance may have some undesirable consequences and its use needs to be regulated. In particular, as is explained below, inheritance tends to undermine encapsulation, it conflicts with the Eiffel's selective export facility, and it may undermine uniformity in a system.

The *conflict between inheritance and encapsulation* is due to the fact that the descendant of a class has free access to its features, and that it can redefine the body of its routines. The potentially negative implications of these aspects of inheritance to encapsulation have been pointed out by Snyder [67].

The *conflict between inheritance and selective export* in Eiffel is due to the fact that anything exported to a class is automatically accessible to all its descendants. To explain why this may be undesirable, consider a class `account` with features `deposit` and `withdraw`. Suppose that in order to ensure that these two routines are used correctly, in conformance with the principle of *double entry accounting*, say, they are exported exclusively to a class `transaction` which is programmed very carefully to observe this principle. Unfortunately, the correctness of the transfer of money in the system may be undermined by any class that inherits from `transaction`, which may be written any time during the process of system development, because any such class would have complete access to the routines `deposit` and `withdraw`.

Finally, the manner in which *inheritance undermines uniformity* can be illustrated as follows: Suppose that we would like *all* accounts in a given system to have precisely the same structure and behavior. This cannot be ensured in the presence of inheritance because, due to polymorphism, instances of any subclass of class `account` can "masquerade" as instances of `account`. Besides having additional features, these "fake"

accounts may have *different behavior* created by redefinition and renaming of features defined in the original class `account`.

For all these reasons one may want to impose constraints on the very ability of a class to inherit from another class, and on the precise relationship between a class and its descendants. In Section 6.2.2 we present the means provided by Darwin-E for imposing constraints over the inheritance graph itself (which in Eiffel can be an arbitrary DAG). In Section 6.2.2 we present the means for restricting the ability of a heir to adapt some of the features it inherits — by *redefinition*, by *renaming* and by *re-export*. In later sections we show how to regulate the accessibility of the various features of a class to the code in its descendants. Finally, in Section 6.2.8, we show how it is possible to *force* certain classes to inherit from certain other classes (see rule $\mathcal{R}40$ in particular).

### Restricting the Ability to Inherit

To regulate the inheritance graph itself we introduce the following interaction:

**Definition 2 (inherit interaction)** *Given two classes* `c1` *and* `c2`*, none of which is the class "*`any`*"*[5]*, we say that the interaction* `inherit(c1,c2)` *occurs if* `c1` *directly inherits from* `c2`*.*

The `cannot_inherit` prohibitions over this interaction can be imposed in a variety of useful ways as illustrated below.

If the law $\mathcal{L}$ contains the following rule:

$\mathcal{R}19$. `cannot_inherit(_,account)`.

then no class would be able to inherit from class `account`, making it a *terminal* class.

If a project is to have many such terminal classes one may mark each of them by the term `terminal`, and include the following rule in $\mathcal{L}$, which would prevent inheritance

[5] *The class* `any` *is a special class in Eiffel, every class implicitly inherits from this class.*

from all such classes.

$\mathcal{R}$20. `cannot_inherit(_,T) :- terminal@T.`

The prohibition of inheritance from a given class may be only partial. For example, the following rule (used alternatively to Rule $\mathcal{R}$19)

$\mathcal{R}$21. `cannot_inherit(C,account) :- not cluster(accounting)@C.`

allows only classes in cluster `accounting` to inherit from class `account`.

A prohibition over inheritance may be only a temporary measure, taken at some stage of the process of system development. For example, suppose that during this process a programmer named John creates a class `c1` which is not yet fully debugged and documented, and therefore cannot be released for general use in the project. Nevertheless, John wants his close collaborator Mary to be able to use this class, in particular by having her own classes inherit from it. This can be done by having John add the following rule to the law of the project:

$\mathcal{R}$22. `cannot_inherit(C,c1) :- not (owner(P)@C, name(mary)@P).`

Finally, the following rule establishes the policy that kernel classes cannot inherit from non-kernel classes, which is necessary for kernelized design.

$\mathcal{R}$23.
```
cannot_inherit(C1,C2) :-
            cluster(kernel)@C1,
            not cluster(kernel)@C2.
```

### Redefinition

In order to regulate redefinition — a well known *necessary evil* of object oriented programming — Darwin-E defines the concept of *redefine interaction* as follows:

**Definition 3 (redefine interaction)** *We say that the interaction* `redefine(c1,f,c2)` *occurs if* `c1` *redefines a feature* `f` *which has been originally defined in class* `c2`*. (By the phrase* `f` *has been "originally defined in* `c2`*" we mean that* `c2` *is the closest ancestor of* `c1` *where* `f` *has been defined, redefined or renamed.)*

Note that the latest version of Eiffel provides some means for regulating this interaction, as follows: one can declare a feature `f` of class `c` to be `frozen`, thus preventing it from being redefined anywhere. This is equivalent to the rule:

$\mathcal{R}24$. `cannot_redefine(_,f,c).`

But the `frozen` specification is, of course, much less expressive then our `cannot_redefine` rules, as demonstrated by the following examples.

Consider the policy that the various features of class `account` cannot be redefined anywhere but by classes that belong to the accounting cluster. This policy can be established by writing the following rule into $\mathcal{L}$.

$\mathcal{R}25$. `cannot_redefine(C,_,account) :- not cluster(accounting)@C.`

As another example, one may want the features defined in kernel classes to have universal semantics, and thus never to be redefined, except, perhaps, by other kernel classes. This policy is established by the following rule:

$\mathcal{R}26$.
```
cannot_redefine(C1,_,C2) :-
            not cluster(kernel)@C1,
            cluster(kernel)@C2.
```

Finally, a purist designer may want to prohibit all redefinition in his system. This policy can be established by means of the following rule.

$\mathcal{R}27$. `cannot_redefine(_,_,_).`

**Renaming**

In Eiffel an inherited feature can be renamed by the heir class. Such renaming may serve two useful purposes: (1) it may help avoiding name clashes, particularly those arising from multiple inheritance; and (2) it may help provide a customized interface to the clients of the heir. But in spite of its usefulness, renaming may sometimes be undesirable, mostly because it reduces uniformity in the system. In order to regulate renaming, Darwin-E defines it as an interaction, as follows:

**Definition 4 (rename interaction)** *We say that the interaction* `rename(c1,f,c2)` *occurs if* `c1` *renames a feature* `f` *which has been originally defined in class* `c2`.

As an example of control over renaming, consider a policy that *exported features of kernel-classes cannot be renamed by non-kernel descendants.* This policy is established by the following rule:

$\mathcal{R}$28.
```
cannot_rename(C1,F,C2) :-
            not cluster(kernel)@C1,
            cluster(kernel)@C2,
            exports(C2,F).
```

The predicate `exports(C2,F)` would invoke a built in predicate which succeeds if the feature `F` is exported, directly or indirectly, from class `C2`.

### Changing the Export Status of Inherited Features

In Eiffel, the export status of a feature `f1` defined in class `c1` can be redefined in any of its descendants. Such a re-export may be undesirable for two reasons. First, any *increase* in the visibility of `f1` may violate the legitimate wishes of the designer of class `c1`. For example, there are good reason to keep the encryption key of a class `encryption` completely hidden. Second, a decrease in the visibility of `f1` would make compile-time type checking impossible, giving rise to a phenomenon called in Eiffel *system-level validity failure* [37]. To provide some control over this capability of Eiffel we introduce the following interaction:

**Definition 5 (changeExp interaction)** *Let* `c1` *be a class,* `f1` *be one of the features defined in* `c1`*, and* `c2` *be a descendant of* `c1`*. We say that the interaction* `changeExp(c2,f1,c1)` *occurs if* `c2` *redefines the export status of* `f1`.

As an example of regulation over this interaction, the builder of the `encryption` class may decide to prohibit any changes in the export status of the feature `key` of this class

by means of the following rule:

$\mathcal{R}$29. `cannot_changeExp(_,key,encryption).`

As another example, a purist system designer may prohibit any change of export status in the system by means of the following rule:

$\mathcal{R}$30. `cannot_changeExp(_,_,_).`

### 6.2.3 Being a Client

A class `c1` is said to be *a client* of class `c2` (the "supplier") if `c1` declares either an attribute, a local variable or a formal parameter of type `c2`. In other words, any use (except for inheritance) of `c2` by `c1` requires `c1` to be a client of `c2`. The client-supplier relation is unconstrained by the Eiffel language, but it sometimes needs to be constrained. For instance, one principle relating to kernelized design may imply that kernel classes should not be clients of non-kernel classes. We therefore define the being-a-client relation as a controllable interaction called "`use`," as follows:

**Definition 6 (use interaction)** *Let* `c1` *and* `c2` *be two (possibly identical) classes. We say that the interaction* `use(c1,c2)` *occurs if* `c1` *is a client of* `c2`.

For example, the following rule:

$\mathcal{R}$31.
```
cannot_use(C1,C2) :-
          cluster(kernel)@C1,
          not cluster(kernel)@C2.
```

prohibits kernel classes from being clients of non-kernel classes.

### 6.2.4 Feature Calls

A feature `f` of an object `x` can be called either remotely, by some other object `y` (using the dot notation `x.f`, with the appropriate arguments, if any), or locally, by object `x` itself. Such calls are constrained in Eiffel by means of the following visibility rules:

1. A feature `f` of class `c` is visible for *local calls* to code written in every descendant of `c` (including, of course, `c` itself).

2. A feature `f` of class `c` is visible for *remote calls* by an object of a class `d`, only if `f` is exported, either universally, or selectively to `d`.

3. Features exported to a class are automatically exported to all its descendants.

Darwin-E subjects both types of feature calls, the remote and the local, to farther regulation. The need for such regulation will become clear in due course.

Since we are committed here to compile-time enforcement of the law, we view a feature call essentially as an interaction between *classes* (parameterized by the features involved) rather than between the objects that are dynamically involved in this interaction. (This fact causes misidentification of the target of the interaction in some rare circumstances, as explained later in this section.) A *call interaction*, is defined as follows:

**Definition 7 (call interaction)** *Consider a feature* `f1` *defined in class* `c1` *and a feature* `f2` *defined in class* `c2` *(see Definition 3 for what we mean by a feature being defined in a given class.) We say that the interaction* `call(f1,c1,f2,c2)` *occurs, if the feature* `f2` *of* `c2` *is called from routine* `f1` *of class* `c1`.

Note that several types of call interactions are excluded from the scope of the `cannot_call` rule-type, as follows:

1. A class calling itself; this is always legal.

2. Calls of creation-procedures as part of the instantiation process (using the `!!` operators). These calls are handled in Section 6.2.5.

3. Calls to most universal features, i.e., those defined in class `any`.(However, calls to the routines `copy`, `deep_copy`, `clone`, and `deep_clone` are controlled by this rule.)

### On the Differences between Exports and `cannot_call` rules

To illustrate the use of `cannot_call` rules, let us return to an example discussed in Section 6.2.2. Consider again the class `account`, and suppose that it defines routines

`deposit`,`withdraw` and `balance`. Consider the following rules.

$\mathcal{R}32$.
```
cannot_call(_,C,F,account) :-
        (F=deposit|F=withdraw),
        not C=transaction.
```

$\mathcal{R}33$.
```
cannot_call(_,C,F,account) :-
        F=balance,
        not cluster(accounting)@C.
```

The first of these rules states that the features `deposit` and `withdraw` cannot be called from anywhere but class `transaction`. This is almost equivalent to having these feature exported selectively to `transaction` — *almost*, but not quite. Features exported selectively to class `transaction` would be usable also by any class that inherits from `transaction`, while under rule $\mathcal{R}32$ class `transaction` would have the *exclusive* power to call these features. Unlike rule $\mathcal{R}32$, rule $\mathcal{R}33$ cannot be even approximated by means of Eiffel export clauses. This rule makes the feature `balance` of class `account` unusable anywhere but in the classes that belong to the `accounting` cluster, whatever they may be. Such a specification, which provides semantics to a cluster of classes, obviously cannot be matched with any construct in Eiffel.

It is important to realize that `cannot_call` rules are prohibitions, not a permission. Thus, for example, for the routine `withdraw` to be actually callable from class `transaction` it must be exported, the Eiffel way, either explicitly to this class, or universally. In general, the potential call-graph (often called as the *may-call* graph) of a system can be obtained by the export clauses of the classes included in it. Under Darwin-E this graph is restricted by our `cannot_call` rules. This gives rise to two approaches to the specification of the may-call graph in a given project. The first is to let the various classes contain their, possibly selective, export clauses, as in a standard Eiffel program, and use our `cannot_call`-rules to specify additional constraints, which would be difficult or impossible to specify by means of export clauses. The second approach is to have all features of a class that are to be exported at all, be exported universally, and rely on the `cannot_call`-rules for the specification of more sophisticated may-call graphs. Both approaches seem reasonable.

### Providing for an Interface of a Cluster

Consider the following policy: *features of any cluster can be called from another cluster only if they are marked as interface_features in the object-base* B. This policy is established by the following rule:

$\mathcal{R}$34.
```
cannot_call(_,C1,F2,C2) :-
            cluster(K1)@C1,
            cluster(K2)@C2,
            K1=/=K2,
            not interface_feature(F2)@C2.
```

### Limitations of Static Analysis of Call Interactions

The static analysis used here to characterize call interactions, may, in some rare circumstances, misidentify the target of the interaction, as follows: Let routine `f1` contain the expression `x.f2`, where `x` is declared to be of class `c2`. This expression would be interpreted as the interaction `call(f1,c1,f2,c2)`. But suppose that dynamically `x` points to an instance of a descendant `c3` of class `c2` which *redefines* `f2`. In this case, the call interaction that actually takes place at runtime is `call(f1,c1,f2,c3)`. This misidentification, which matters only if the law makes different rulings about these two interactions, can be removed by means of run-time analysis. But this is hardly worthwhile due to the rarity of the potential error.

## 6.2.5 Generation of Objects

New objects are generated in an Eiffel program mostly by means of the instantiation operator `!!`, but also by *cloning*. Both of these are recognized in Darwin-E as instances the `generate` interactions defined below:

**Definition 8 (generate interaction)** *Let* `c1` *be a class, and* `r1` *a routine defined in it. We say that the interaction* `generate(r1,c1,v2,c2)` *occurs if routine* `r1` *contains an expression that, if carried out, would generate a new object of class* `c2` *into variable* `v2`. *A generate interaction is identified as such if routine* `r1` *has an expression that has one of the following forms:*

1. `!!v2.p`, *when* `v2` *is of class* `c2` *with* `p` *as a creation routine.*

2. `!c2!v2.p`, *where* `v2` *is declared to be of a subclass of* `c2` *and* `p` *is a valid creation routine for* `c2`[6].

3. `v2 := clone(v3)`, *where both* `v2` *and* `v3` *are of class* `c2`.

To illustrate the use of control over the generation of objects, consider the policy which requires that accounts, i.e., instances of class `account`, be created only by means of classes in the accounting cluster. One interpretation of this policy is established by the following rule:

```
R35. cannot_generate(_,C1,_,account) :-
            not cluster(accounting)@C1.
```

This rule prevents class `account` from being instantiated anywhere but in the accounting cluster. Note, however, that this rule says nothing about the instantiation of descendants of class `account`, which in a strong sense constitute the creation of accounts too. To include these in our policy we replace rule $\mathcal{R}35$ with the following rule:

```
R36. cannot_generate(_,C1,_,C2) :-
            not cluster(accounting)@C1
            heirOf(C2,account).
```

where `heirOf(C,D)` invokes a built in rule of Darwin-E which succeeds when class `C` is a descendant of `D`, or is equal to it.

Note that unlike most other languages, the latest version of Eiffel does provide some means for regulating the ability to generate instances of a given class, by selective export of its *creation routines*. But the target of selective export is specified *by extension*, with all the limitation of such specification described in Section 6.2.4. Indeed, none of the example policies above can be established in Eiffel itself.

Note also that the examples used in this section do not use the first and third parameter of the `generate` interaction. The use of these parameters will be illustrated in examples presented in later chapters.

---

[6] *Note that if the class* `c2` *does not have any creators part, then the creation construct may not have the creation-call part. We describe only the most general form of creation constructs.*

### A Limitation of the Static Analysis of generate Interaction

Using a combination of *polymorphic assignment, cloning* and *reverse assignment* it is possible to thwart our `cannot_generate` prohibitions. But the offending combination can be prevented by other means, as we shall see.

The problem at hand is demonstrated below. Suppose that law $\mathcal{L}$ contains Rule $\mathcal{R}35$ above which permits only classes in cluster `accounting` to generate accounts. A class `c` not in cluster accounting can nevertheless generate accounts as follows.

Suppose that `c` has the entities `x1` and `x2` of class account, and `a1` and `a2` of class `any`; and let `x1` point to an actual account generated elsewhere. The following sequence of instructions in class `c` would generate a new account pointed to by `x2`.

```
(1)   a1 := x1;
(2)   a2 := clone(a1);
(3)   x2 ?= a1;
```

Statement (1) stores a pointer to the original account object in variable `a1` of class `any`. Since there is no prohibition in $\mathcal{L}$ against generating objects of class `any`, `a1` can be cloned (statement (2)) into `a2`. Now `a2` contains a pointer to a new account. So far no real harm is done, because variable `a2` cannot be really used *as an account*. However, statement (3) reverse-assigns this object to the account variable `x2`, making it into an official, usable account.

Such violations of `cannot_generate` rules can be averted by preventing reverse assignment into instances of class `account`, using `cannot_revAssign` rules discussed in Section 6.2.7.

Eiffel provides yet another means for the violation of `cannot_generate` rules, namely the `deep_clone` routine, which may generate a great variety of objects in a single call. We view this routine as one of the unsafe features of the Eiffel language, which should be tightly regulated, by means of `cannot_call` rules, thus reducing its danger to any prohibitions over generation of objects that the law may contain.

### 6.2.6 Assignment

Assignments are already very restricted in Eiffel, which does not allow any cross object assignment. But additional constraints on assignment may be useful for several reasons. In particular, for ensuring that some types of objects are *immutable*, that certain functions do not produce side effects, and for eliminating cross class assignment in certain circumstances, as we shall see below. For these, and some other reasons we treat assignment as a controllable interaction, as defined below:

**Definition 9 (assign interaction)** *Let* `r1` *be a routine defined in class* `c1`*, and let* `a2` *be an attribute defined in an ancestor* `c2` *of* `c1` *(*`c2` *may, in particular, be equal to* `c1`*). We say that the interaction* `assign(r1,c1,a2,c2)` *occurs if the code of routine* `r1` *has a statement that assigns (or reverse-assigns) into* `a2`*.*

Here are some elaborations on this definition:

1. Three kinds of statements are covered by this interaction: (a) the *assignment statement* $a2$ := ...; (b) the *reverse assignment* $a2?$ = ...; and (c) the *creation statement* !!$a2$. The latter case is considered an assignment because it stores in $a2$ a pointer to the newly created object. (Note that creation is also controlled independently by means of the `generate` interaction discussed above.)

2. Only assignment to *attributes* is covered by this interaction, not assignment to local variables of a routine.

3. This interaction *does not* cover assignment made by routines declared as creation routines. The reason for this exemption, which removes creation routines from any potential prohibition over assignment, is that assignment is the *raison d'etre* of these routines.

4. The parameter `c2` of the interaction `assign(r1,c1,a2,c2)` refers to the class in which attribute `a2` is defined, *not* the class of this attribute. This is because the nature of control we envision over assignment, which will be illustrated later on in this section.

Note also that if one wants to restrict the ability to change the value of an attribute of an object `x`, one should worry about a copy operation `x.copy(...)`, which in effect

assigns to all the attribute of `x`. We, therefore, view this operation as a kind of assign interaction, as defined below:

**Definition 10 ((another) assign interaction)** *Let* `r1` *be a routine defined in class* `c1`*, and let* `x` *be a variable of class* `c2`*. We say that the interaction* `assign(r1,c1,_,c2)` *occurs if the code of routine* `r1` *has a statement* `x.copy(...)` *or* `x.deep_copy(...)`.[7] *(Note that the third argument of this interaction is the variable (in the sense of Prolog) which means, in effect, that it effects all attributes of class* `c2`*.)*

Note that the `copy` case of a call interaction is independently subject to the static analysis error of call interactions, as discussed in 6.2.4.

The control provided by Darwin-E over the assign interaction has several important applications. One application is discussed below; additional applications are presented in later chapters.

### Fortifying Encapsulation in Eiffel

One of the controversial design decisions of Eiffel is to provide an heir class with complete access to all the features it inherits. While this may simplify the code in the heir class, it compromises the encapsulation provided by the parent classes, in the general manner discussed in [67]. We can fortify encapsulation in Eiffel, without giving up much of the ease of access provided by it, by allowing a heir only read access to the attributes it inherits. Rule $\mathcal{R}37$ below accomplishes this by allowing an assignment to be carried out only by a class on the attributes defined in it.

$\mathcal{R}37$. `cannot_assign(_,C1,_,C2) :- C1=/=C2.`

This rule would prevent any assignment to a variable defined in a class `C2` by code written in any proper descended of it, while not disturbing the read and call access provided by Eiffel.

---

[7] *(Of course the operation* `x.deep_copy(...)` *may do more then assign into its explicit target* `x`*, and its precise range cannot be determined at compile time. This, however, is a rarely used operation whose use can be tightly regulated separately by means of* `cannot_call` *rule.*

### 6.2.7 Reverse Assignment

*Reverse assignment* is a type-safe means provided by Eiffel[37] to "resurrect" a pointer stored in variable of more general type then the object being pointed to, making this object usable for what it really is. This, somewhat unusual device, which is very useful for polymorphic and strongly typed languages, is used in the following manner: Let `x1` be a variable of class `c1`, and let `x2` be a variable of class `c2`, which is a descended of `c1`. The reverse assignment statement `x2 ?= x1` would store in `x2` the pointer to the object pointed to by `x1`, if this pointer happens to be of class `c2`; otherwise, the value `void` is stored in `x2`.

Although reverse assignment can be regulated in Darwin-E as a special case of assignment, by means of `cannot_assign` rules, there are reasons to provide a regulation mechanism specific to it. One such reason is that if used carelessly reverse assignment can make variables void, and thus cause run-time exceptions. Furthermore, reverse assignment can be used to foil some of the controls provided by Darwin-E, as has been discussed in Section 6.2.5. We therefore define the following interaction:

**Definition 11 (revAssign interaction)** *Let* `r1` *be a routine defined in class* `c1`*, and let* `a2` *be an attribute of class* `c2`*. We say that the interaction* `revAssign(r1,c1,a2,c2)` *occurs if the code of routine* `r1` *has a statement of the form* `a2 ?= ...;`

As an example, recall Rule $\mathcal{R}35$ in Section 6.2.5, intended to establish the policy that accounts cannot be created anywhere but in classes of the accounting cluster. As discussed in Section 6.2.5 this policy can be foiled with a use of reverse assignment. The offending use of reverse assignment can be blocked, however, by means of the following rule:

```
R38. cannot_revAssign(_,C1,_,account) :-
            not cluster(accounting)@C1.
```

which prevents reverse assignment into variable of class `account` by any class outside of the accounting cluster.

### 6.2.8 Inclusion of a Class in a Configuration

Darwin-E regulates the very inclusion of classes in configurations via what we call the `include` interaction, defined below.

**Definition 12 (include interaction)** *Recall that a configuration object in Darwin-E represent a collection of classes to be assembled together to form a runnable system. Now, given a class* `c` *and a configuration* `g`*, we say that the interaction* `include(c,g)` *occurs if* `c` *is included in* `g`*.*

We provide here two examples of control over this interaction. First, suppose that configurations marked by the term `release` are intended for actual release to the customer, and that classes marked by term `tested` have been officially tested (recall that under Darwin-E it is possible to control who can mark a given class as tested). The following rule establishes the policy that release-configurations can include only tested classes.

$\mathcal{R}$39.
```
cannot_include(C,G) :-
            release@G,
            not tested@C.
```

For our second example, consider a class called `inspection` built in such a way that it allows the inspection of all component parts of instances of all its descendants. (This should be possible if we allow class `inspection` to use C-code.) Now, suppose that we want everything defined in cluster `accounting` to be inspectable in this way, (providing for a degree of on-line auditing of accounting). This can be accomplished by means of the following rule, which *forces all accounting classes to inherit from class* `inspection`.

$\mathcal{R}$40.
```
cannot_include(C,_) :-
            cluster(accounting)@C,
            not inherits(inspection)@C.
```

## 6.3 Cost of Imposing Regularity

In this section we will discuss the cost of imposing regularity on an Eiffel system. To be specific, we are mostly interested in the perceivable manifestation of the complexity: the time elapsed between start and completion of activities resulted from an `enforce` command. If a builder is allowed by the law of the project to issue such a command, the following two major steps of activities begin:

1. Static Analysis of the code of the system at hand to obtain the interactions, and

2. Enforcement of the law of the project upon these interactions.

The static analysis starts with an intermediate-form representation of all user-defined classes included in a system. This intermediate-form is a by-product of computing the built-in interface-related properties of a module-object. Recall that the interface-related properties are obtained by pre-processing the code when it is being attached[8] to a module-object. Since the cost for computing this intermediate-form will not show up in the perceived speed of our `enforcement` command, we will not be concerned about the pre-processing step anymore.

Given this, the static analysis step requires a single pass over the intermediate-form representations of all user-defined classes in the configuration at hand. This step is independent upon the law of the system and therefore the time spent on this can be estimated as $O(LOC_u \times |S|)$ in the worst case, where $|S|$ denotes the number of classes in the system $S$ at hand, including library classes and $LOC_u$ is the number of lines of code in the user-defined classes in $S$. This is so because each source-code line may generate at most a constant number of construct in the intermediate-form representation, and for each such constructs, the static analysis phase may have to look up not only the user-defined classes, but also any library classes, for feature definition, type or class information to extract the corresponding interaction.

The cost of the second step depends on the nature of rules users define to express regularity. Things get complicated if such rules involve recursion and function symbols. We will therefore, estimate the cost for this step in terms of rule invocations, with the understanding that the actual cost will be bounded by this estimate, times the worst case cost of evaluating a user-defined rule. In Darwin-E we have 10 kinds of controllable interactions. In a system $S$ let $N_i$, $1 \leq i \leq 10$, denote the number of interactions of the $i$th kind. Let there be $M_i$, $1 \leq i \leq 10$, number of Darwin-E rules in the law of

[8]Since a module-object `o` is re-attached each time its code is modified, this pre-processing, which is a single pass over the code attached to `o`, will probably be carried out many times during the life of `o`. However, this pre-processing is independent of the number of configurations that may include `o`. In particular, even if a module-object representing an user-defined class is never included in any system, its code will still be pre-processed upon attachment.

$S$ for controlling the $i$th kind of interactions. Then checking whether or not a single interaction of $i$th kind violates the law of $S$ may take $M_i$ rule invocations. Therefore checking all the $i$th interactions is $M_i \times N_i$ rule invocations. The total estimated number of rule invocations for checking interactions of all kinds is $\sum_{i=1}^{10}(M_i \times N_i)$. Note that $\sum_{i=1}^{10} M_i$ is the number of rules in the law of a project, of the order of 100s with each $M_i$ being of the order of 10s. Each $N_i$ however is a considerably large and $\sum_{i=1}^{10} N_i$ is $O(LOC_u)$.

Clearly, this second step is very expensive, and we made every effort to minimize the $N_i$ s in our environment (refer to Appendix B for details). We would like to investigate alternative approaches such as storing the $N_i$ interactions in a relational database and the querying it to find out if there is any violation. The advantage then will be the availability of various query optimization techniques at our disposal, but translating regularities into such queries, may be tricky, and even impossible at times, a matter which is yet to be investigated.

# Chapter 7

# Legislation in Darwin-E

By the term *legislation* we mean the process of changing the law of the system by creation and destruction of rules, and the manner in which this process itself is regulated by the law. We start our discussion of legislation in Darwin-E with a description of rule-objects and their effect on the law.

## 7.1 Rule-Objects and their Effect on the Law

Recall that the law of the project, $\mathcal{L}$, is a Prolog program consisting of a collection of clauses contributed by various rule-objects in the object-base. The contribution to the law of each rule-object is the clause

```
head:- body
```

contained in property

```
#ruleForm((head:-body))
```

of the object. For example, the contribution to the law of the following rule-object:

```
#id(r1)
#type(rule)
#legislator(jack)
#ruleForm((
           cannot_useC(Class,_):-
                   not cluster(kernel)@Class
cluster(kernel)
          ))
```

is the clause:

```
cannot_useC(Class,_):- not cluster(kernel)@Class.
```

A point of caution: being a Prolog system, the law $\mathcal{L}$ is a *sequence* of rules, and the order of rules in this sequence may matter. In Darwin-E, the order of rules in $\mathcal{L}$ is the order of the creation of rule-objects that contributes these rules. However, for a wide range of applications it is possible to formulate the user-defined rules in Darwin-E in such a way that their order, and thus their creation order, does not matter. The law of the example project presented in Chapter 10 is an example of such order independent set of rules.

But such an order independent formulation of rules require some care, as is illustrated in the following situation. Consider, for example, a rule with rule-form

```
canDo(S,run,configX):- $do(set(executed)@configX)
```

created after (the creation of) a rule whose rule-form is the general permission

```
canDo(S,X,T):- true.
```

In this case, a

```
^run => configX
```

message will always be allowed by the general permission, and the user-defined property `executed` in `configX` can never be set as intended.

Finally, let us use the above example rule-object to describe its various properties. The primitive properties `#id(r1)` stores the identity `r1` of the rule-object and the primitive property `#type(rule)` distinguishes it as a rule-object in the object-base. The property `#ruleForm((_))` is mentioned earlier. The property `#legislator(jack)`, stores the identity of the builder (`jack`) who created the rule-object[1]. User-defined properties may also be present in rule-objects to represent project-specific facts: the property `cluster(kernel)` in the above rule is one such, denoting the fact that it is a kernel rule (i.e. belongs to the kernel cluster). Such properties can be useful in enforcing project-specific legislative constraints, such as, *only a kernel-programmer can remove kernel rules.*(Refer to Section 7.7.2 for an example.)

[1] We will often use the term *legislator* to refer to the creator of a rule, for instance, we could say that `jack` is the legislator of `r1`.

## 7.2 Creation of Rule-Objects — an Overview

New rule-objects are created in Darwin-E by means of `^createRule(...)` messages sent to *metarules*. A metarule is a special kind of object that serves as a template for rule creation, allowing for the creation of a certain kind of rules. Of course, different `^createRule(...)` messages sent to a given metarule `mr` would create different rule-objects, depending on the parameters of this message, and on the nature of `mr`. The set of all rules creatable from `mr` by all such messages is called the *extent* of this metarule, or, symbolically, $extent(mr)$.

The creation of rule-objects is controlled by two orthogonal devices:

1. The very existence of some metarules in the object-base, providing the potential for certain kinds of rules to be created.

2. The control over who can send which `^createRule(...)` messages to which metarules. (Such control is provided by appropriate `canDo` rules.)

The Darwin-E environment provides several basic metarules, which can be farther customized for the initial state of any given project, as we shall see.

## 7.3 The Structure of Metarules

The following is an abridged textual representation of a simple metarule:

```
#id(canDoMr)
#type(metarule)
#legislator(#creator)
#ruleSchema((
    canDo(Builder,Operation,Object):- #delta
            ))
#nono([$do(create(A)@B),$do(destroy@C),.....])
```

The primitive properties `#id(canDoMr)`, `#type(metarule)` needs no explanation. As in rule-objects, the (primitive) property `#legislator(#creator)` denotes that this metarule was created by the (built-in) builder `#creator`. The primitive property `#nono([...])` is used to store the nono-list of a metarule, which is a list of Prolog terms and Darwin-E primitives. The significance of the nono-list is discussed later.

In the above example the nono-list, shown only in part, contains various Darwin-E primitives of the form `$do(...)`. The primitive property `#ruleSchema(...)` stores the rule-schema of the metarule which, again, has the form `head:- body` of a Prolog clause. The rule-schema serves as the template of the rule-forms of the rules creatable from the metarule. The `head` could be any valid Prolog term, in our example it is the Prolog goal

```
canDo(Builder,Operation,Object).
```

The `body` of the rule-schema may contain Prolog terms including LGA primitives defined in Darwin-E of the form `$do(_)` and condition tests of the form `_@_`. One of the terms in the `body` must be a special term `#delta`. The special variable `#legislator` may appear in any term in `body`. The significance of `#delta` and `#legislator` will be explained later. In our example the `body` consists of the singleton term `#delta`.

## 7.4 The Rule Creation Mechanism

Consider a metarule `mr` whose rule-schema is $h_m$ `:-` $b_m$ and a message

```
^createRule(r1,(h:- b))
```

sent to `mr` by builder `x`. Since the most fundamental part of rule creation is derivation of the rule-form from the rule-schema of the parent metarule, we will first explain how the rule-form $h_r$ `:-` $b_r$ of the new rule `r1` is produced.

- **(Deriving the head)** The head $h_r$ of the new rule-form is derived by unifying `h` and $h_m$. This involves unification of the arguments (if any) in `h` and $h_m$. If $h_m$ and `h` do not unify the rule creation process is aborted without creating any new rule.

- **(Deriving the body)** The $b_m$ in the rule-schema of the metarule is transformed into the body $b_r$ of the new rule-form through the following steps:

- First, `b` is checked against the nono-list of `mr`. If it contains any term included in the nono-list then no new rule is created and the rule creation process terminates, otherwise it proceeds to the next step.
- If the term `#legislator` appears anywhere in $b_m$, it is *replaced* by the identity of the builder `x` who sent the `^createRule` message. If no such term appears in $b_m$ no action is performed at this step.
- If `b` involves arguments that were affected during earlier unification of $h_m$ and `h`, these arguments are substituted by the appropriate unifiers to form `b*`, and then the term `#delta` in $b_m$ is *replaced* by this `b*`. If `b` does not involve such arguments then `#delta` is simply replaced by `b`.
- If $b_m$ involves arguments that were affected during earlier unifications then the unification is carried over to these arguments also.

The substitution of the of the term `#legislator` and the check against the nono-list are crucial. We will have more to say about these later.

The new rule will have the `#type(rule)` and `#id(r1)` primitive properties. The new rule-form $h_r$ `:-` $b_r$ will be stored as the `#ruleForm((`$h_r$ `:-` $b_r$`))` property. The identity of the legislator will be stored as `#legislator(x)`. In addition `r1` will have all user defined properties present in `mr`.

We will now demonstrate the rule creation activity with an example. Only the built-in builder `#creator` has the ability to create new rules initially, so let us assume that during the initialization of a project, `#creator` sends the following message:

```
^createRule(projectRule1,(
            canDo(S,new(X),builder):-
                                role(manager)@S
                                              ))
                                             => canDoMr.
```

This creates the following rule that allows the project manager to create new builders:

```
#id(projectRule1)
#type(rule)
#legislator(#creator)
#ruleForm((canDo(S,new(X),builder):-
                                   role(manager)@S
          ))
```

The above example illustrates the following facts about rule creation activity. First, the variables `Operation` and `Object` in the rule-schema of `canDoMr` is unified with `new(X)` and `builder` respectively. Second, as a result of the *delta replacement* the new rule-form gets the `role(manager)@S` term in its body. Finally, that the `#creator` created this rule is noted in the `#legislator(#creator)` property.

## 7.5 Specializing Metarules by Creating New Ones

A metarule can be customized to suit the requirement at hand by creating new metarules out of it. New metarules strengthen the constraints expressed by the rule-schema of the parent and may contain more user-defined properties than the parent.

From a given metarule we can have a hierarchy of metarules. It is possible to build the hierarchy in such a way that the extents of the sibling metarules govern distinct aspects of the project. This way legislation of the rules that govern these aspects can be controlled independently[2].

A new meatrule `mr1` is created from an existing metarule `mr` by sending a

```
^createMetarule(mr1,(h:-b),n)
```

message to `mr`. The mechanism of metarule creation is similar to that of rule creation, but there are certain differences. We will discuss the differences and then give an example. First, there are three arguments in the message involved metarule creation as opposed to the two involved in rule creation. The additional argument `n` in this case is actually a list (possibly empty) of Prolog terms and Darwin-E directives that is *appended* to the nono-list of `mr` to compute the nono-list of `mr1`. Second, the `b` in the

```
^createMetarule(mr1,(h:-b),n)
```

message must contain the special term `#delta`. This is required so that after replacing the `#delta` in the rule-schema of `mr`, we still have a `#delta` in the rule-schema of `mr1`.

[2]This principle is widely used in Darwin-E in formulating the built-in metarules that are visible to the users. We have already mentioned such a metarule `canDoMr`, for a complete list of these metarules refer to appendix E. Users can build their own hierarchy of customized metarules starting from any of these built-in metarules.

Like rules, creation of metarules is also controlled by two orthogonal devices: 1) the very existence of the parent metarule and 2) the ability to send the `^createMeatrule(...)` message. The following example will illustrate various issues in metarule creation.

Let us consider the effect of sending the following message to the metarule `canDoMr` that we introduced earlier:

```
^createMetarule(canDoMr0,(
                    canDo(S,M,T):- cluster(C)@T,
                                   cluster(C)@S,
                                   #delta), []) => canDoMr.
```

Let us also assume that the law approves of sending such a message. The metarule `canDoMr0` whose rule-schema is presented below, will be created as result.

```
canDoMr0:    canDo(S,M,T):- cluster(C)@T,
                            cluster(C)@S,
                            #delta.
```

The following facts about metarule creation are to be noted. First, the head of the rule-schema of the child metarule must unify with the head of the rule-schema of the parent. Therefore, the head of the rule-schema of a metarule determines what goals will be defined by the rules created from it or its child metarules. For example, only rules of the form `canDo(S,M,T):- ...` can be created from the `canDoMr` metarule or its descendant metarules such as the `canDoMr0` presented above. Second, the body of the rule-schema of the child contains the terms other than `#delta` in the body of the rule-schema of the parent, subject to any substitution of variables resulted from earlier unifications. Thus, the rule-schema of the child metarule can express a stronger constraint than the rule-schema of the parent. For example, a rule `r` in the extent of `canDoMr0` concerns only to builders and classes that belong to the same cluster, whereas rules in the extent of `canDoMr` can regulate the activity of any builder on any class. Third, because of the fact that the nono list of the parent is included in the child's nono list, the terms and directives allowed in rules creatable from the child can appear in the rules creatable from the parent, but not the reverse. Finally, as a result of all these, note that the extent of the child metarule is included in the extent of the parent metarule: a rule that could be created from the child metarule, can also be

created from the parent metarule. However, a rule creatable from the parent may not be possible to create from the child because the child may have a restricted rule-schema and extended nono-list.

## 7.6 Controlling The Scope and Power of Rules

We characterize the scope of a rule by the objects under its jurisdiction. Often times it is needed that the scope of a rule should depend upon the legislator. For example, consider the requirement that the activities in a cluster is subject to the permission of its supervisor, and the supervisor can permit activities that relate only to his cluster. This means that the scope of the `canDo` rules a supervisor writes should be limited to his cluster. The special variable `#legislator` in the rule-schema of a metarule can be used for this, as explained below.

Consider the rule-schema of a new metarule `canDoMr1`:

```
canDoMr1:    canDo(S,M,T):- cluster(C)@ #legislator,
                            cluster(C)@T,
                            cluster(C)@S,
                            #delta.
```

Recall that the `#legislator` term in the rule-schema will be substituted by the identity of the legislator during rule creation.

Let us assume that the supervisors are allowed to create rules from this new metarule. In that case, if the supervisor of the kernel cluster (let us assume that the builder `jack` is the supervisor of the kernel cluster), decides to allow any kernel programmer to set properties (recall that builders cannot set built-in properties) on kernel objects, he just needs to send the following message:

```
^createRule(ra,(
            canDo(S,set(_),T):- true
                                                    )) => canDoMr0.
```

This will create the rule $\mathcal{R}41$ shown below:

```
R41. canDo(S,set(_),T) :-
            cluster(C)@jack,cluster(C)@S,
            cluster(C)@T.
```

Note the result of substituting the `#legislator` variable in the above rule: it can only permit builders in `jack`'s cluster (which is the kernel cluster) to set properties in object belonging to the same cluster.

The power of a rule is characterized by the operations it performs in its scope. This power depends on the `$do(_@_)` terms that appear in the body of its rule-form and in the bodies of the rule-form(s) of any other rule(s) being called (recursively) from it. A metarule `mr` partly [3] determines the power of the rules in *extent*(*mr*). The `$do(_@_)` terms in its rule-schema, subject to unification and variable substitution during rule creation, are inherited in the rule-form of the rules in *extent*(*mr*). What other terms can appear in the rule-form of such a rule is further controlled by the nono list `L`, stored as the built in property `#nono(L)` of `mr`: the nono list of a metarule `mr` specifies which terms cannot appear in the rule-form of any rule in *extent*(*mr*). The nono list of a metarule is included in the nono list of all its child metarules. Therefore, the terms prevented by the nono list of `mr`, can never appear in the rules in *extent*(*mr*).

The use of the nono list is illustrated in the following example. Recall that a `canDo` rule can be used to prescribe additional operations in addition to permitting builders to use Darwin-E commands. We have seen examples of such `canDo` rules earlier in Section 5.1. Let us consider the metarule `canDoMr1` presented above, from which the supervisors of different clusters will create `canDo` rules to permit activities within their respective clusters. In order to have some control over the supervisors, we may decide that the supervisors should not be allowed to prescribe additional operations in the `canDo` rules they create. This can be achieved by including the term `[$do(_@_)]` in the nono list of `canDoMr1`.

---

[3] Partly, because there is a way to by-pass the control exerted by a metarule on the power of the rules in its extent. But we will not go into that details here.

## 7.7 Controlling the Use of Legislative Commands by canDo Rules

So far, we have discussed the role of metarules in the process of rule or metarule creation, assuming that law permits the legislators issue the appropriate legislative commands. In Darwin-E we have three legislative commands, two of them are already introduced in the context of rule and metarule creation. In the following Section, we will show examples of `canDo` rules that control the use of these commands. The third command will be introduced in Section 7.7.2.

### 7.7.1 Example canDo Rules Controlling Creation of Rules and MetaRules

Let us consider a project, in which only the project manager is allowed to create new metarules from the built-in `canDoMr` metarule. The following rule defines the necessary permission:

$\mathcal{R}$42. `canDo(S,createMeatrule(_,_,_), canDoMr) :- role(manager)@S.`

Let us assume that the manager has created the new metarule `canDoMr0` introduced earlier, with the intention that the supervisors of each cluster will create `canDo` rules from it, and the `canDo` rules thus created will belong to the cluster of their respective legislator. The following rule grants the cluster supervisors the appropriate power:

$\mathcal{R}$43.
```
canDo(S,createRule(X,_), canDoMr0) :-
            role(supervisor)@S,
            cluster(C)@S,
            $do(set(cluster(C))@X).
```

### 7.7.2 Destruction of Rules and Metarules

Darwin-E provides a single command `removeRule` to destroy rules and metarules. Destruction of a rule or a meatrule means its removal from Darwin-E s object-base. In the law we can specify who can send the `^removeRule` message to which rule or metarule and thus control the destruction of rules and metarules from the law.

For instance, kernel programmers may be allowed to destroy any rule or metarule that belongs to the kernel cluster by the following rule:

```
R44. canDo(S,removeRule,T) :-
            cluster(kernel)@S,
            cluster(kernel)@T.
```

As another example, consider the following rule:

```
R45. canDo(S,removeRule,T) :- #legislator(S)@T.
```

Recall that the primitive property `#legslator(X)` stores the identity of the builder who created the rule as `X`. Therefore, the above rule allows a builder to destroy **only** the rules he created.

### 7.7.3 Legislative Control and Evolutionary Invariants

To conclude the discussion on legislation we point out an important ramification of legislative control in relation to the notion of evolutionary invariants. Over the lifetime of a project, the *law*, as conceived at the outset of the project, is expected to evolve. This will be reflected in the project by continuous addition of new rules and metarules and removal of existing ones. This activity, as we have discussed earlier, is governed by the law itself.

Recall that we would like to impose desired regularities as invariant of evolution. Since, regularities are formulated as Darwin-E rules, the legislative structure should be carefully designed to keep this aspect in mind. In particular attention should be paid in formulating what rules can be created and destroyed: a rule that violates an invariance should not be created and a rule that establishes such an invariance should not be destroyed.

# Chapter 8

# Configuration and Version Management in Darwin-E

It is impossible to avoid different incarnations of individual software components (classes in Darwin-E) in a large project. This is a consequence of different forces, such as changes in requirements, changes in the underlying data representation and the algorithm, or testing and bug fixing. Usually, a change does not result in destroying the existing incarnation. Rather, multiple incarnations are kept alive, so that they can suitably be used if needed. We will refer to such different incarnations of a single class as its different versions.

The presence of different versions of classes in a project gives rise to two issues: a) how to manage different versions of a class in the object-base of a project and b) how they affect the task of building a configuration. A software development environment is expected to address both the issues. Various version and configuration management schemes are already in use [4, 60, 71]. In this chapter we describe the configuration and version management facilities of Darwin-E.

Let us mention at the very beginning, that we do not have a full fledged version control and configuration management system in the prototype. Darwin-E provides a minimal support for managing versions with an emphasis on how to use different versions during configuration binding. Various aspects relevant for version control and configuration management are either not considered at all, or supported in a simplistic manner. For instance, representation and extraction of changes in version à la RCS [71] is not implemented, locks and transactions are implemented only minimally.

## 8.1 Version and Groups

In Darwin-E, a group-object collectively represents different versions of a class. Existence of the group for a given class named `c` is not mandatory and a module-object `o`, representing a particular version of `c`, may be enrolled in its group or it may exist in the object-base by itself. A module-object is enrolled in its group by a Darwin-E command, which like all Darwin-E commands is issued as messages, and governed by the law of the project. This law-governed control is the basis of the flexible version management mechanism of Darwin-E. But first, let us explain the issues involving the creation of new versions of an individual class.

### 8.1.1 Creation and Representation of Versions

To create a new version of a class `c`, a new module-object `o`, designated to represent class `c`, has to be created, followed by the attachment of code to it. This code can either be developed from scratch or can be copied from an existing version[1] of the same class and modifying it. Note that unlike RCS[71] or SCCS [60], Darwin-E does not store and manipulates *delta*s (changes) by itself: versions are stored in its entirety and are to be done explicitly by users.

As an aside, it is also possible to modify the code attached to a module-object `o` *in place*, without creating a new version by invoking `emacs` as follows:

```
john:  ^emacs ⇒ o.
```

However, when emacs quits, all the configurations that may have contained `o` will be notified that `o` has become *dirty*, which signifies that such configurations needs to update the code of `o` before they could be compiled or run.

Creation of versions are controlled by `canDo` rules that control creation and attachment of module-objects, such rules are already discussed in Section 5.3.

---

[1] To copy the code attached to a class object `o` into a file `f`, a builder `john` may use the `cp` command as shown in the message:

```
john:  ^cp(f) ⇒ o.
```

When code is being attached to a newly created module-object or the code attached to an existing module-object is being edited, it may be necessary to hide the intermediary state of the object from others. This can be done by acquiring a lock on the object in question (Locks are discussed in Section 5.2).

### 8.1.2 Associating a Version With Its Group

A typical user `john` can associate a module-object `o` representing some version of a class `c` with a group `g` by issuing the `enroll` command as the following message:

```
john:  ^enroll(o) ⇒ g.
```

If the law of the project permits `john` to issue this command and the group `g` is meant for representing versions of class `c`, then `o` is included in the group `g`. As a result of such enrollment, `g` notes `o` as the most recent member (denoted by the (built-in) property `current(o)`) and the date and time of its enrollment (denoted by another (built-in) property `engrouped(o,timestamp)`, where `timestamp` is the current date and time as obtained from Prolog). If `o` represents a different class than `g`'s class, `o` is not included in `g` and a message informing the builder who issued the `enroll` command about the mismatch is generated. Similarly, `john` can remove a module-object `o` from a group `g` by issuing the `withdraw` command as the following message:

```
john:  ^withdraw(o) ⇒ g.
```

If the law of the project permits him to issue this command, and `o` in fact belongs to `g`, `o` is removed from `g`. If `o` happens to be the current member of `g`, the version that was enrolled just before `o` was enrolled is reinstated as the current member. If `o` is not a member of `g`, no action is taken.

Control over associating versions to groups in a project can be achieved by creating individual `canDo` rules:

```
R46. canDo(S,M,G) :-
            (M= enroll(O) | M= withdraw(O)),
            owner(S)@O.
```

Rule $\mathcal{R}46$ allows only the owner of a module-object to enroll it in its group or remove it from the group. Alternatively, the rule $\mathcal{R}47$ can be used to allow only the owner of the group to do the job:

```
R47. canDo(S,M,G) :-
            (M= enroll(O) | M= withdraw(O)),
            owner(S)@G.
```

The `enroll` and `withdraw` commands belong to the `modifierOp` category. Figure 8.1 describes the use of the built-in goal `modifierOp(_)` in this context in a manner similar to the filtering scheme described in Section 5.3.

```
R48. canDo(S,M,T) :- modifierOp(M),
            canModify(S,M,T).
```

*Commands in* `modifierOp` *category are controlled by* `canModify(_,_,_)` *rules. There can be many* `canDo` *rules to filter different categories of commands.*

```
R49. canModify(S,M,T) :- type(group)@T,
            canModifyGroup(S,M,T).
```

*Commands that are targeted at groups are controlled by* `canModifyGroup(_,_,_)` *rules. There can be different* `canModify` *rules similar to this, to filter modifier commands.*

```
R50. canModifyGroup(S,enroll(O),T) :-
            role(manager)@S,tested@O.
```

*The manager can enroll tested modules. There can be different* `canModifyGroup` *rules to handle different modifier commands affecting group objects.*

Figure 8.1: Filtering `enroll` and `withdraw` from `modiferOp` category

### 8.1.3 Organizing Versions Enrolled In a Group

The default organization of modules enrolled in a group is based upon the time-stamp associated with the module-objects when they are enrolled. But often, a different

organization is needed. For instance, they may be organized based upon some notion of version numbers. As another example, the `versionOf` and `revisionOf` relations may be used to capture the multi-dimensional line of evolution in organizing the group. Although Darwin-E does not provide any command for organizing classes enrolled in a group, it is possible to establish such different organizations by means of various user-defined properties, as described below.

### Organization Based on version Numbers

Version numbers can be associated with module-objects enrolled in a group by means of the property `versionNumber(_)`, which can be used (we will see examples in Section 8.2.2 ) to select a specific version during configuration binding.

To control the manipulation of this property, the following rule may be included in the law of a project that utilizes the filtering scheme described in Section 5.3:

$\mathcal{R}$51.
```
canModifyClass(S,M,T) :-
        (M= set(versionNumber(_)) | M= recant(versionNumber(_))),
        belongsTo(T)@G,
        owner(S)@G.
```

*Owner of a group can manipulate version numbers of classes that belong to his group. The property* `belongsTo(o)` *in a group* `g` *is set by Darwin-E during enrollment of* `o` *in* `g`.

Alternatively, the `versionNumber(_)` property can be set automatically as a side effect when a class is being enrolled in a group. The rule $\mathcal{R}$50 of Figure 8.1 can be modified as shown below to effect this:

$\mathcal{R}$52.
```
canModifyGroup(S,enroll(O),G) :-
        role(manager)@S, tested@O
        $do(recant(currentVn(N))@G),
        M is N + 1,
        $do(set(currentVn(M))@G),
        $do(set(versionNumber(N))@O).
```

*A manager can enroll a tested class (module object) in a group as in Rule $\mathcal{R}$50, but the class is given a version number as well.*

This rule assumes that the property `currentVn(_)` is properly initiated in group objects, which could easily be done by including the Darwin-E primitive `$do(set(currentVn(1))@T)`

in the rule permitting creation of groups. An example of such a rule ( $\mathcal{R}14$) is shown in Section 5.3.

The above rule prescribes setting of a property in a module-object from a `canModifyGroup` rule, which is supposed to handle modifier commands targeted at group-objects. Although this happens as a side-effect, this may be not always be acceptable. This apparent conflict can easily be removed if we use `versionNumber(o,n)` property in a group to denote that `o`'s version number is `n`:

$\mathcal{R}53$.
```
canModifyGroup(S,enroll(O),G) :-
        role(manager)@S, tested@O
        $do(recant(currentVn(N))@G),
        M is N + 1,
        $do(set(currentVn(M))@G),
        $do(set(versionNumber(O,N))@G).
```

*Same as Rule $\mathcal{R}52$, but the version information of a class is stored in the group it belongs to.*

### Organization Based on `versionOf` and `revisionOf` Relations

The version numbers in the previous section (rules $\mathcal{R}52$, $\mathcal{R}53$) are decimal numbers, which means the conventionally used version numbers such as `version(3.1.3.2b)`, denoting evolution along multiple branches, cannot be used. However, using the `versionOf` and `revisionOf` relations, multi-dimensional branching, such as the one shown in Figure 8.2, can be represented.

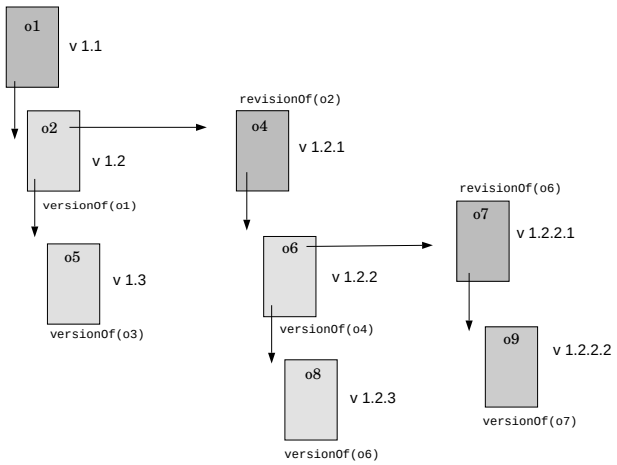


Figure 8.2: Schematic Representation of `versionOf` and `revisionOf` relations

Figure 8.2 depicts an organization of classes in 2-D plane (typical version numbers using a conventional scheme, for this particular example are also shown beside the objects). The organization is represented by the `versionOf` and `revisionOf` relations. As shown in the figure, each such relation is established by means of properties in the module-objects. For example, `o2` is denoted as a version of `o1`, by the property `versionOf(o1)` in `o2`. Similarly, `o3` is denoted as a revision of `o2`, by the property `revisionOf(o2)`. We are mainly concerned here about the representation of the relations. The criteria for being a version or a revision is beyond the scope of our laws and that is why it is left unspecified. However, we acknowledge that establishing such a relation is critical, and should be controlled. In Darwin-E, this can be achieved by controlling the manipulation of the `versionOf(_)` and `revisionOf(_)` properties. As an example, consider the following rule which is very similar to Rule $\mathcal{R}51$:

$\mathcal{R}54$.
```
canModifyClass(S,M,O) :-
        (M= set(versionOf(X)) | M= set(revisionOf(X))),
        belongsTo(O)@G,
        owner(S)@G,
        owner(S1)@X,
        approvedBy(O,S1).
```

*Owner of a group can set the* `versionOf(x)` *or* `revisionOf(x)` *properties in a class object* `o` *that belongs to his group, only if the owner of* `x` *has approved* `o`. *The approval, in this example is in the form of the* `approvedBy` *rules, called as a subgoal of the form* `approvedBy(_,_)`.

## 8.2 Building a Configuration

In Darwin-E, an eiffel system $\mathcal{S}$ is built as a configuration $\mathcal{C}$ by including module-objects representing user-defined classes[2] in $\mathcal{C}$, which results in copying the associated code into the directory associated with $\mathcal{C}$. Usually, there is a choice involved in building a configuration, because of the presence of multiple versions of a single user-defined class (Note that, on the other hand, there is usually only one version of each class in one installation of the Eiffel library.) User-defined classes can be put in a configuration

[2]Classes from installation library that are needed in a configuration, are included automatically by Darwin-E. Users need not worry about them.

in two ways: a) the module-objects that represent these classes can be manually chosen and put explicitly in the configuration or b) a consensus-based configuration binding process can be activated. The command `put(L)`, where `L` can be a list of module objects, group objects or a heterogeneous list of both, is used in an overloaded manner in this regard.

Note that the `include` interactions, introduced in Section 6.2, are a direct consequence of choosing specific versions of classes for a configuration, either explicitly or by configuration binding. As we shall see in this section, the very choice of versions for a configuration can be constrained. Therefore, the classes constituting a system are subject to constraints at two levels: a) choice of specific versions of a class are constrained by various (selection) criteria during configuration build, and b) whether the classes included are acceptable or not, as defined by the `can_include` or `cannot_include` rules in the system sub-law. The latter kind of constraints are related to the structure of the system at hand, and can be useful in expressing various regularities.

### 8.2.1 Inducting Specific Classes Into Configurations

If the list `L` in the `put(L)` command contains a module-object `o`, then we are explicitly selecting `o` to be included in a configuration. For example, the message

```
john:  ^put([o1,o2,o3]) ⇒ cfg,
```

can be used by `john` to put three specific module-objects `o1,o2` and `o3` into the configuration `cfg`, provided of course, that the law allows him to issue such a command. By sending another message

```
john:  ^put([o4,o5,...])  ⇒ cfg,
```

he can add more classes into `cfg`. Attempts to put a class twice (for example, including `o1` in the second `put` command as well) in a configuration is detected by Darwin-E and is not allowed.

After the classes are put into a configuration this way, to notify Darwin-E about the root class of the system, `john` needs to send the following message:

```
john:  ^root => cfg,
```

which will prompt him to enter the name of the root class. As a result of this command, the Ace file needed by the underlying Eiffel implementation ( ISE Eiffel 3.1), is created and customized. Note that the configuration at this point would be ready to compile under normal circumstances, but Darwin-E requires that the (system sub-)law of the project be enforced upon it before compilation.

Building configurations in this manner is controlled by defining who can put which classes in which configuration. The following rule allows the owner of the configuration to put classes into it:

$\mathcal{R}$55. `canDo(S,put(L),CFG) :- owner(S)@CFG.`

Since `put` is a `configurationOp` command, if the refinement convention of Section 5.3 is used in the law, control over the `put` command can be established by the `canConfigure` rules:

$\mathcal{R}$56.
```
canConfigure(S,put(L),CFG) :- owner(S)@CFG,
        not ( in(O,L), not tested@O).
```

*The owner of the configuration can put only the tested classes into his configuration.*

When we are putting specific classes into a configuration, it is sometimes convenient and even necessary, to specify a set of classes that must be included. Such an intention can be stated by means of the `must_include` rules:

$\mathcal{R}$57.
```
must_include(CFG,[o1,o2,o3,o4]) :-
            className(toolkit)@o1,
            className(window)@o2,
            className(scroll_bar)@o3,
            className(button)@o4,
            cluster(gui)@CFG.
```

*This rule forces any configuration in the* `gui` *cluster to include the classes* `toolkit`, `button`, `window` *and* `scroll_bar`.

As an aside, let us note that, with the help of the legislative facilities, it is possible to make sure that a `must_include` rule created by a builder `b` applies only to the configurations owned by `b`.

Control over inducting specific classes into a configuration, as shown in the above examples, can be interesting and complicated enough. However, the case of consensus based configuration binding, which we will discuss next, is more natural in a cooperative environment.

### 8.2.2 Consensus Based Configuration Binding

In a large project, the idea of hand-picking specific module-objects for each configuration is often impractical, because of the large number of classes in the object-base. It is more so in a cooperative environment, where the configuration builder is supposed to use classes developed by many different developers. Apart from his requirements, he has to abide by the concerns of the developers and corporate policies that may apply, while choosing classes to build his configuration.

We have group-objects in Darwin-E to represent and organize versions of classes. Since a configuration builder knows the names of the classes he wants to induct in his configuration, he can build the configuration in terms of the groups that represent these classes, leaving the selection of specific classes for each such group to be done later by what is known as a *configuration binding* process. In Darwin-E we provide a consensus based configuration binding mechanism (which is invoked by sending the `bind` command to the configuration in question) that takes into account the concerns of various parties involved.

In the current implementation of Darwin-E, the binding mechanism tries to reach a *consensus* among the constraints expressed by the three parties: the configuration builder, the class developer and management. It succeeds only if, for each group inducted in a configuration, Darwin-E can find a version that satisfies the applicable constraints.

The mood of the constraints set forth by the three parties, however, are not same: a configuration builder *requests* the inclusion of the classes of his desire, a class developer *approves* the inclusion of his classes to certain configurations and management

*prohibits* certain selections. Furthermore, because of their very nature, a notion of ownership (denoted by the `owner(_)` property in our rules) is implicitly associated with the constraints of the configuration builders and class developers: a configuration builder expresses concerns about *his* configurations, a class developer sets forth constraints about inclusion of *his* classes. Constraints imposed by management, on the other hand are of a project-wide nature, and hence does not need to be implicitly associated with the notion of ownership.

In the following three sections we will describe each of these constraints in detail, followed by an example showing how all these can be put to work. Note that this particular three-party-consensus is just one example framework that we have chosen for Darwin-E (because of its simplicity and the natural mapping of the three players (configuration builder, class developer and management) involved in binding), the law-governed paradigm can be used to construct many such frameworks as discussed earlier in [53].

### Concerns of The Configuration Builder

A configuration builder usually has certain criteria by which he prefers one version over another, to bind to a group he inducted in his configuration. Typically, these criteria are stated in terms of how the versions are organized in the group, and other associated properties of the versions. Darwin-E provides a means for expressing such concerns by means of `requestInclusion` rules. For example, a typical builder `john` may intend to have only the current version (recall that the last class object enrolled in a group is designated as its current version) in all the configurations he owns, which can be

expressed by the following rule:

```
R58. requestInclusion(CFG,G,V) :- owner(john)@CFG,
            current(V)@G.
```

If he wishes to consider only the versions that were enrolled in a group in the year 1995, he would create the following rule:

```
R59. requestInclusion(CFG,G,V) :- owner(john)@CFG,
            engrouped(V,date(X,_,_,_,_,_)@G,
            X = 95.
```

*Note that* `date(yy,mo,dd,hh,mm,ss)` *is the format in which the timestamp is obtained from Prolog by Darwin-E.*

The constraints expressed by rules $\mathcal{R}58$ and $\mathcal{R}59$ uses the default organization of versions in a group, namely the timestamp ordering and the designation of the currently enrolled version. The typical builder may formulate his requirements in terms of any other organization imposed upon the members of the group. For example, if a version number is assigned to group members, he might specify his intention to consider versions of a certain number and higher for his configuration:

```
R60. requestInclusion(CFG,G,V) :- owner(john)@CFG,
            versionNumber(V,N)@G,
            N >= 3.
```

`john` *is looking for classes of version 3 or up for his configurations.*

The rules and constraints expressed by them discussed so far, apply to *all* groups inducted in *all* the configurations owned by a typical builder `john`. If desired, he may formulate rules that apply to only some of the group she inducted and/or some of his configurations. For example, in the Figure 8.3, we formulate a situation where `john` has different criteria for different configurations as well as different groups. The rules in Figure 8.3 use various properties associated with classes and groups such as `tested` or `release(_)`, in addition to properties like `current(_)`.

Finally, let us show how the `versionOf` and `revisionOf` relations, used for organizing the members of a group, can be used in `requestInclusion` rules. But first, let

$\mathcal{R}$61. `requestInclusion(CFG,G,V) :- owner(john)@CFG,`
`release(external)@CFG,`
`includeInExtRelease(CFG,G,V).`

*The* `release(external)` *property of a configuration designates it for external release. For such configurations,* `john` *uses the* `includeInExtRelease` *rules to formulate his constraints.*

$\mathcal{R}$62. `requestInclusion(CFG,G,V) :- owner(john)@CFG,`
`release(internal)@CFG.`

*The* `release(internal)` *property of a configuration designates it for internal release. For such configurations,* `john` *does not have any specific constraints, that is, any version is acceptable.*

$\mathcal{R}$63. `includeInExtRelease(CFG,G,V) :- in(G,[g1,g2,g3,g4,g5,g6]),`
`current(V)@G,`
`tested@V.`

*The* `includeInExtRelease` *rules treat the groups* `g1,g2,g3,g4,g5` *and* `g6` *differently. The class objects selected from these groups should be tested as well as current versions of the corresponding groups.*

$\mathcal{R}$64. `includeInExtRelease(CFG,G,V) :- not in(G,[g1,g2,g3,g4,g5,g6]),`
`tested@V.`

*For other groups, they need only be tested.*

Figure 8.3: Expressing different Inclusion Criteria for Different Configurations and Different Groups

us define the `isVersionOf(x,y)` goal, that succeeds if `x` is a version of `y`, by means of the following rules:

$\mathcal{R}$62. `isVersionOf(X,Y) :- versionOf(Y)@X.`

$\mathcal{R}$63.
```
isVersionOf(X,Y) :- versionOf(Z)@X,
            isVersionOf(Z,Y).
```

We needed this to capture the transitivity of the `versionOf` relation, which is not reflected in the `versionOf(_)` properties of module objects. The above two rules will be auxiliary to `requestInclusion`(and `approveInclusion` and `managerObjection` rules to be introduced later) rules using the `versionOf` relation.

In order to specify that for a group `g`, `john` is going to accept versions of a particular revision (i.e. a vertical branch in Figure 8.2), we would need a rule like the one presented below:

$\mathcal{R}$64.
```
requestInclusion(CFG,g,V) :- owner(john)@CFG,
            isVersionOf(V,o4).
```

*Any version of a specific object* `o4` *(i.e. in Figure 8.2* `o6,o8,..`*) is acceptable for* `g` *in any of* `john`*'s configurations.*

As another example, let us consider the following rule:

$\mathcal{R}$65.
```
requestInclusion(CFG,G,V) :- owner(john)@CFG,
            isVersionOf(V,X),
            revisionOf(Y)@X,
            owner(john)@Y.
```

Although, it looks complicated, this rule expresses a simple constraint: any version of a revision owned by himself is acceptable for any group in any of `john`'s configurations.

### Concerns of The Class Developer

As the implementor of a class, a builder may approve the inclusion of her classes only in certain configurations. In Darwin-E, a builder can express such approvals by means of the `approveInclusion` rules. For example, a typical builder `jill` may decide that

the versions she owns can only be used in her configurations. The rule she needs to create for that is as follows:

$\mathcal{R}66$.
```
approveInclusion(CFG,G,V) :- owner(jill)@CFG,
        owner(jill)@V.
```

The antithesis of this rather selfish constraint is to approve the use of her classes anywhere, which can be expressed by the following rule:

$\mathcal{R}67$.
```
approveInclusion(CFG,G,V) :- owner(jill)@V.
```

Both the above rules apply to versions owned by `jill` enrolled in *any* group being considered for inclusion in *any* configuration. The universal quantification in either places is not a necessity. If she wishes, `jill` can make her approval less liberal, as shown by the following rule:

$\mathcal{R}68$.
```
approveInclusion(CFG,G,V) :- owner(jill)@V,
        owner(S)@CFG,
        in(S,[jill,patty,linda]).
```

> `jill` *approves inclusion of her classes in the configurations owned by the ones listed (The sex of the selection of configuration builders prefered by* `jill` *is purely accidental.) in the rule.*

Alternatively, she can reduce the scope of her approval to only some of the classes she owns (assuming version numbers are used in the project):

$\mathcal{R}69$.
```
approveInclusion(CFG,G,V) :- owner(jill)@V,
        versionNumber(V,N)@G,
        N <= 3.
```

> *Any of her classes with version numbers 3 or less can be freely included in any configuration.*

As a final example, consider the following approval, applicable to a project where `versionOf` and `revisionOf` relations are used to organize group members.

$\mathcal{R}70$.
```
approveInclusion(CFG,G,V) :- owner(jill)@V,
        isVersionOf(V,X),
        revisionOf(Y)@X,
        approveInclusion(CFG,G,Y)
```

This rule recursively calls itself with a different parameter set. By this rule, `jill` approves inclusion of her class `v` in a configuration `cfg`, if there is an approval to

include in `cfg` the revision `r`, of which `v` is a version.

### Management Policies

To model the management policies about configuration binding we have implemented the `managerObjection` rules. Such rules essentially *prohibit* certain versions from being selected for certain configurations. For example, the management may consider classes that were enrolled in a group before a certain date to be unfit for a configuration meant for external release. This policy can be expressed and enforced by the following rule:

$\mathcal{R}$71. `managerObjection(CFG,G,V) :- release(external)@CFG,`
`engrouped(V,date(YY,_,_,_,_,_)@G,`
`YY < 95.`

*Classes enrolled in their group in 1995 will no longer be considered for inclusion in configurations meant for external release.*

As another example, consider the management policy that untested classes owned by trainee programmers cannot be included in any configuration. The rule to implement this policy is shown below:

$\mathcal{R}$72. `managerObjection(CFG,G,V) :- owner(S)@V,`
`role(trainee)@S,`
`not tested@V.`

As a final example, consider the management decision that in certain configurations (designated by the `pureEiffel` property), no class that uses external (C language) routines can be used. This can be established by the following rule:

$\mathcal{R}$73. `managerObjection(CFG,G,V) :- pureEiffel@CFG,`
`external(_,language(c))@V.`

*Note that Darwin-E sets the property* `external(r,language(c))` *in a class object only if it has an external (written in C) routine* `r`.

### An Example Scenario

If the consensus based binding mechanism is to be used in a project, the law of the project must allow builders to introduce `requestInclusion` and `approveInclusion`

rules. All such rules cannot be part of the initial law: new objects (builders, classes, groups and configurations), possibly involving new properties, are expected to be created during the life of the project and it is impossible to foresee all these and formulate the `requestInclusion` and `approveInclusion` rules accordingly. Similarly, the managers should be allowed to introduce `managerObjection` rules.

Before initiating the binding process, the desired groups must be inducted in the configuration in question. The presence of `requestInclusion` and `approveInclusion` rules is mandatory in order to launch the binding mechanism, however if there is no managerial constraints, `managerObjection` rules need not be created.

Let us consider a project, which at a certain time `t` has the following `requestInclusion`, `approveInclusion` and `managerObjection` rules, as shown in Figure 8.4 in its law.

```
R74. requestInclusion(CFG,G,V) :- owner(john)@CFG.


R75. approveInclusion(CFG,G,V) :- owner(john)@V,
            owner(john)@CFG.

R76. approveInclusion(CFG,G,V) :- owner(jill)@V.

R77. managerObjection(CFG,G,V) :- owner(john)@CFG,
            current(V)@G.
```

Figure 8.4: Consensus Based Configuration Binding: An Example

At this point `john` has inducted 2 groups `g1` and `g2` in his configuration `cfg` and has initiated the binding process by issuing the following message:

```
john:  ^bind => cfg,
```

which is approved by the law. Let us assume that versions `o1, o2, o3` are enrolled in `g1` and the versions `o4` and `o5` are enrolled in `g2`. The owner of `o1` and `o4` is `john`, the owner of `o2` and `o5` is `jill` and `o3` is owned by `linda`. Let us also assume that `o3` and `o5` are current versions of `g1` and `g2` respectively, and `o2` was enrolled in `g1` after `o1`.

The binding mechanism considers the versions enrolled in a group in FIFO order. So, `o3` will be considered for `g1` first. This choice passes the `requestInclusion` rules since Rule $\mathcal{R}74$ accepts any choice. But, there is no `approveInclusion` rule to approve

o3 to be included in john's configuration. So, o2 will be considered next. This choice passes the requestInclusion, approveInclusion and managerObjection rules. Then the binding mechanism will try to choose a version for g2 in a similar manner. In this case o5 will be considered first, this choice passes the requestInclusion and approveInclusion rules, but Rule $\mathcal{R}77$ prohibits current versions to be included in john's configurations. So the binding mechanism will try again with o4 and this choice will satisfy all the constraints. As a result of the binding o2 will be chosen for g1 and o4 for g2. If however, linda was the owner of o4 rather than john, the binding would fail, despite the successful binding of previous groups (i.e. g1 with o2).

## 8.3 Comparision With Other Approaches

What we just described is a framework as implemented in Darwin-E, necessary for the consensus based configuration binding mechanism described in [53]. To establish the novelty of our work, we will now contrast our approach with several others.

Unlike RCS [71] or SCCS [60] we do not provide any change management. Rather, we assume that designating a version or revision, or organizing multiple incarnations of a class is done by users. However, such actions are regulated by law. We showed that versions can be organized in many different ways by using various properties and, that such properties can be used in a novel configuration binding mechanism. RCS or SCCS does not support configuration binding and compilation management, so they are normally used in conjunction with systems like Make [15].

Make [15] (and its subsequent extensions) is one of the earliest and most successful systems for compilation management. It does not provide any selection mechanism at all. A similar compilation management system (ISE Eiffel 3 compilation manager) is used by Darwin-E, when a bound configuration is being compiled.

DSEE [29, 30] is another file based tool, like RCS,SCS or Make, but it has a richer functionality than any of these. DSEE can represent descent lines and variant branches like RCS/SCCS, and in addition it provides a limited selection mechanism. Selection criteria can be specified by means of *configuration threads* and typical examples of

selection criteria are 'choose the ones that I reserved' or ' choose the ones that are changed'.

The next set of systems attempt to avoid the dependency on files as the units they manage and uses the higher order notion of modules. In Cedar [70], a module is an interface-implementation pair, each implementation can have different versions. Modules are identified by unique names and versions are identified by the date of creation. The selection of versions can only be guided by their date of creation. It works for cedar language only and is capable to represent one line of development.

Gandalf [20] extends the Cedar notion of a module by allowing each version to consist of a set of revisions. Thus it can represent a two dimensional line of development. Selection of the appropriate realization of an interface of a module is by allocating a default, which can be changed dynamically.

Adele [4] further extends the notion of a module by allowing a set of views to define an interface, which can be realized by different variants, and each variant has different revisions. Selection criteria of Adele configuration manager uses properties much like we utilize properties in Darwin-E.

Systems like ClearCase [28] attempt to address an additional set of issues involving version control and configuration management, namely distributed and concurrent build, which we do not address at all.

The Darwin-E groups are essentially used in the same way other configuration management systems use the interfaces of modules: to facilitate selection of the appropriate realization. The selection mechanism is often trivialized in existing systems; except for Adele no one has used the properties to guide the selection. However, they only have a fixed set of properties to use in this context, where as we can use practically any user defined properties. Furthermore, no other system has attempted to model the interplay of different constraints set forth by different parties regarding configuration binding. In fact, the consensus framework presented here owes its very existence to the law-governed paradigm. The basic idea is fairly language independent, it is possible to implement a similar mechanism involving groups and different relations such as `versionOf`, `revisionOf` etc. over classes (modules) written in other languages. On

the other hand, we have trivialized many version control issues: storage, representation and management of changes, visibility of change, answering queries regarding versions and configurations etc.

# Chapter 9

# Monitoring Facilities of Darwin-E

In [40], Minsky introduced the notion of an *auditable software project*. Such a project calls for monitoring the activities of the builders, and independent on-line monitoring of the systems they build. LGA provides a suitable base for realizing these, as evidenced by the monitoring facilities of Darwin-E.

## 9.1 Monitoring Builder Activities

By monitoring builder activities in a project we mean one or both of the following:

- Immediate notification of the activity, and
- Archiving a record of the activity which could later be reviewed.

We would like to be able to specify what gets monitored, as well as who is privy to the monitored information. In the following two sections, we describe how this is achieved in Darwin-E, along with several demonstrative examples. The rules presented in these examples utilizes various user-defined properties like `role(_)`, `cluster(_)` etc. that were introduced earlier.

### 9.1.1 Immediate Notification

Darwin-E provides two primitives `$do(ucast(message)@b)` and `$do(bcast(mesg))`, where `mesg` is a Prolog term and `b` is a builder, whose effects are as follows. The primitive `$do(ucast(mesg)@b)` results in displaying the term `mesg` along with current time and date in the screen of builder `b`, provided builder `b` is active (i.e. has an active window associated with it, connected as a client of the project server). The primitive

`$do(bcast(mesg))` causes display of the term `mesg` in the screens of all active builders in the project[1]. Such primitives can be included in the body of a `canDo` rule $r$ in order to send notifications of the events permitted by $r$, when they occur. The following examples elaborate:

$\mathcal{R}75$. `canDo(S,M,T) :- modifierOp(M), cluster(kernel)@T,`
`cluster(kernel)@S,role(manager)@X,`
`$do(ucast(message(S,M,T))@X).`

*Note the use of Prolog variable* `X`*, that binds to a builder that has the* `role(manager)` *attribute.*

The rule $\mathcal{R}75$ allows kernel programmers to modify kernel classes by means of modifier commands, but at the same time it makes sure that a manager is notified about such an activity.

As another example, consider the following rule:

$\mathcal{R}76$. `canDo(S,M,T) :- creationOp(M,O), cluster(kernel)@S,`
`$do(set(cluster(kernel))@O),`
`$do(bcast(message(S,M,T))).`

The goal `creationOp(M,O)` in rule $\mathcal{R}76$ above, is defined by a built-in Darwin-E rule that succeeds when `M` is bound to a creation command, and returns the object being created in `O`. Rule $\mathcal{R}76$ permits kernel programmers to create kernel objects, and at the same time makes sure that such an event is reported to all active builders. In this case, all active builders are designated as the monitors of creation activities in the kernel cluster.

### 9.1.2 Archiving Message Sending Events

Darwin-E provides the primitive `$do(d_monitor(s,m,t))`, which stores the information that `s` issued the command `m` to `t`, in the built-in Darwin-E object `#tracer`. This primitive, when included in a `canDo` rule $r$, results in storing trace information about

[1]Note that `bcast` does not imply conventional *broadcast*, what happens here is that messages are sent out to the individual builders (that are active currently) one after another.

all messages approved by $r$. The following examples demonstrate:

```
R77. canDo(S,M,T) :-
            role(trainee)@S,
            trainee_permission(S,M,T),
            $do(d_monitor(S,M,T)).
```

Rule $\mathcal{R}77$ invokes an auxiliary goal `trainee_permission(_,_,_)`, (to be defined by a rule created by some other authority) which defines what a trainee can do. However, rule $\mathcal{R}77$ guarantees that a trace of activities of all trainees will be archived, no matter what kind of `trainee_permission` rules are actually present.

As another example, consider the following rule:

```
R78. canDo(S,M,T) :- cluster(base)@S,cluster(base)@T,
            b_b(S,M,T),
            (monitor_b_b(S,M,T) → $do(trace(S,M,T)) | true).
```

*The* → *is* if *in some implementations of Prolog:* `c → a| b` *means, if* `c` *then* `a` *else* `b`.

The implication of rule $\mathcal{R}78$ on the activities of the base programmers on the objects belonging to their own cluster is twofold: a) which such activities are allowed is defined by the `b_b` rules, and b) which of these allowed activities are monitored are defined by the `monitor_b_b` rules. Notice the separation of permission and monitoring: which activities will be monitored can be specified dynamically as and when needed by creating (and deleting) appropriate `monitor_b_b` rules independent of the `b_b` rules that define the power of the base programmers. Since it is possible in Darwin-E to control rule creation, we may stipulate that only the manager can create the `monitor_b_b` rules. This, in turn, gives only the managers the power to monitor the activities concerning base programmers.

### Manipulating The Archived Information

A builder `john` can view the archived information about a message

```
s:  ^m ⇒ t,
```

by issuing the command `showTrace(s,m,t)` to `#tracer`, provided the law approves `john` to use this command. If there is a record about such a message in `#tracer` it will be displayed on the screen of `john`. If there are multiple records i.e. the same message was sent many times, all such records will be displayed.

The arguments of the `showTrace` commands can be Prolog variables as well, for example, `john` may send the following message:

```
john:  ^showTrace(jill,M,c) ⇒ #tracer.
```

If the law approves this message, any command `jill` may have issued to `c` ( because the Prolog variable `M` can unify with any such command ) that was traced, will be displayed. In order to view all the commands that `jill` may have issued to any object, `john` would send the message:

```
john:  ^showTrace(jill,M,T) ⇒ #tracer.
```

The command for deleting the trace information `remTrace(_,_,_)` works in a similar fashion. In order to delete the records, if any, about the message

```
s:  ^m ⇒ t,
```

`john` would send the `remTrace(s,m,t)` command to `#tracer`. If the law approves such an activity by `john`, records about this message will be deleted from `#tracer`. Similarly, in order to delete all the trace information about `jill`, `john` would send the following message:

```
john:  ^remTrace(jill,M,T) ⇒ #tracer.
```

It is possible to control the use of `showTrace(_,_,_)` and `remTrace(_,_,_)` commands. Darwin-E provides a built-in rule that defines a goal `traceOp(X,Sender,Message,Target)`, which succeeds if X is either `showTrace(s,m,t)` or `remTrace(s,m,t)` and binds `Sender` to `s`, `Message` to `m` and `Target` to `t` as a result. This goal can be used in `canDo` rules

to selectively permit manipulation of the trace information stored in `#tracer`. The following examples elaborate:

```
R79. canDo(S,M,T) :- traceOp(M,jill,_,_),
            role(manager)@S.
```

The above rule gives the manager to manipulated any trace information about `jill`'s activities.

```
R80. canDo(S,M,T) :- traceOp(M,_,_,Target),
            role(manager)@S,
            cluster(kernel)@Target.
```

The above rule allows the manager to manipulate trace information about commands invoked on kernel objects.

### 9.1.3 On the Use of the Notification and Archiving Primitives

Although the outcome will be different, the three Darwin-E primitives `$do(ucast(message)@b)`, `$do(bcast(message))` and `$do(trace(s,m,t))`, can be used interchangeably in `canDo` rules. In order to avoid repetition, we showed their usage in different types of formulation in rules $\mathcal{R}75$ through $\mathcal{R}78$, however the primitive shown in any of these rules can be replaced by one or more primitives from this set. For example, the rule $\mathcal{R}78$ can be reformulated as below:

```
R81. canDo(S,M,T) :- cluster(base)@S,cluster(base)@T,
            b_b(S,M,T),
            (
            monitor_b_b(S,M,T) →
            (
            $do(trace(S,M,T)),
            role(manager)@X,
            $do(ucast(b_b_activity(S,M,T))@X)
            ) | true
            ).
```

In this case the activities that satisfy `monitor_b_b` rules, will be recorded in `#tracer` and in addition, such an activity will also be reported to a manager immediately.

Finally, such primitives can also be included in any auxiliary rule that is being invoked from a `canDo` rule, such as the `canModify` or `canModifyGroup` rules introduced earlier. Therefore, monitoring can seamlessly blend into the structured control over builder activities based on command categories.

### 9.1.4 Related Work

Almost all software development environments incorporate some means to trace user activity. In the Marvel [5] software development environment, forward chaining of rules that model the software process can trigger tracing and notification. There are dedicated event-driven announcement tools such as YEAST [27], which monitors the occurrence of events in order to trigger various notification activities. An example of using tools like YEAST in a process-oriented environment can be found in Provence [26]. Our approach does not incorporate chaining: one event cannot generate a chain of notification and trace activities; and the novelty of our approach lies in the use of law, which provides a simple and flexible means to specify which events will be monitored.

## 9.2 On-Line Monitoring of Eiffel Systems

By *on-line monitoring* we mean observing and analyzing selected computational events of a given system $\mathcal{S}$ as and when they occur. On-line monitoring techniques have been used for various tasks ranging from debugging and testing, to performance analysis and fault-tolerance [64].

In an object-oriented computation, objects interact among themselves via feature calls, so our primary focus in monitoring an object-oriented system is the call interaction. In addition to call, we also monitor the generation interaction, since in some applications, object generation can be critical (such as generation of `money` or `account` in a financial system) and may require monitoring.

To facilitate such on-line monitoring, one needs to embed certain *observers* or *sensors* in $\mathcal{S}$. In order to make sure that the monitoring activities do not degrade the performance of $\mathcal{S}$, these sensors often just report the events to an object outside $\mathcal{S}$ or

running a different thread of computation, responsible for analyzing the reported events and taking follow-up actions.

We would like to be able to specify which (call and generation) interactions will be monitored, and then embed the sensors in $\mathcal{S}$ accordingly, without any knowledge of the programmers who built $\mathcal{S}$. In this section we describe how this has been done in Darwin-E.

### 9.2.1 Implementation Details

On-line monitoring of a system $\mathcal{S}$ in a project under Darwin-E can be summarized as follows. Immediately after a monitored interaction takes place, details about the interaction, including the identity of the two objects involved, are reported to a special purpose object called spy. This object, equipped to handle such reporting, is guaranteed to exist when $\mathcal{S}$ is run. Which interactions will be monitored are specified by the law (the system sub-law, to be precise) of the project and the code instrumentation required for this kind of reporting is performed by Darwin-E when the law is enforced upon $\mathcal{S}$[2]. The spy object can be customized in projects to perform the required analysis and follow-up actions, either by itself or in conjunction with other objects as the case may be.

We will describe this implementation in three parts: first, we will show how the required code instrumentation is arranged from the law. Then we will show how the special object spy is made available to the system at run time. Finally we will describe how to use these facilities in a project.

[2]This assumes that the source code is available to us. In other words the monitoring scheme described here will not work if the source code is unavailable.

### Code Instrumentation Arranged by Law

Darwin-E provides the primitive `$do_monitor(_,_,_,_))` that can be included in a `can_call` or a `can_generate` rule as follows:

```
R82. can_call(F1,C1,F2,C2) :- cluster(kernel)@C2,
            not cluster(kernel)@C1,
            $do(monitor(F1,C1,F2,C2)).

R83. can_generate(F1,C1,V2,C2) :- cluster(accounting)@C1,
            heirOf(C2,account),
            $do(monitor(F1,C1,V2,C2)).
```

The effect of the primitive in Rule $\mathcal{R}82$, which permits non-kernel classes to call kernel classes, is described below. For each call `o.f(l)` in a non kernel class, where the declared type of `o` is a kernel class `c` , Darwin-E will insert the call

```
spy.c_monitor(current, o,'o','c','f','l');
```

after the construct that contains `o.f(l)`. The first argument `current` of `c_monitor` refers to the entity which performed the call, the second argument represents the entity `o` to which the call was made. The third argument is a string denoting the identity of the callee. The fourth argument denotes the identity of the class to which the call was made as understood by Darwin-E[3]. The fifth and sixth arguments, as can be seen, are the name of the procedure being called and the argument-list of the call.

The effect of the primitive in Rule $\mathcal{R}83$, that permits classes in the accounting cluster to generate instances of `account` and its descendants, is slightly different. After each instruction that leads to generation of such an entity `o`, it inserts the call

```
spy.g_monitor(current, o,'o','c','f','l');
```

where `f` is the creation routine of class `c` that is being used. The 6 arguments of `g_monitor` have the same meaning and purpose as those of `c_monitor`.

---

[3]Notice that `c` is the target of the call `o.f(l)` as understood by Darwin-E, which means it is a module object representing class `c`, and the feature `f` was defined in `c`. However at runtime `o` may not be an instance of `c`, (dynamic binding and subtype polymorphism), the dynamic type of the callee entity can be derived using Eiffel language features in the class of `spy` or its cohorts.

The call to spy via c_monitor or f_monitor routines are the sensors that report the details of an interaction to spy. The $do(monitor(_,_,_,_)) primitive in a rule $r$ instruments the code of a system in such a way that such a sensor is attached to the interactions permitted by $r$. This is how the interactions to be monitored are specified by the law. Note that the code instrumentation being done by Darwin-E and no cooperation of the developers of the system being monitored is needed in this matter of incorporating sensors.

Let us now point out that instrumenting source code for monitoring presents an inherent difficulty for function calls, a difficulty that does not arise in case of procedure calls. A procedure p can only be called as the last call in a call chain like a.b....c.d.p. Thus the monitoring code can be inserted after the call construct and therefore the desired monitoring will be done right after the call we want to monitor. This, however is not the case with functions. A function f may be called in the middle of a call chain. There is no easy way to automatically insert the monitoring code right after the call via f; the only reasonable place to insert the monitoring code seems to be at the end of the call chain. This way we run into the risk that monitoring will occur at an indeterminate time after the call took place[4]; and by then the state of the system may have changed and we may end up monitoring an undesired state of the system. Therefore, we suggest that one should mostly monitor the procedures calls, rather than the function calls. This is not a serious limitation, in part, because it is generally accepted that functions should not change the state of objects (query versus command), and we can in fact impose such a restriction on functions (we will see more about side-effect-free (SEF) functions later in Chapter 11) under Darwin-E.

### Incorporating The Special Object spy

In order to ensure that there is a single spy object, able to accept calls via g_monitor and c_monitor, available at run time, Darwin-E provides two Eiffel classes (described below) SPYLOCATOR and SPYMASTER, which are to be used as follows.

---

[4]In the extreme case, the program may terminate via some exception before the control returns to the end of the call chain.

```
class SPYLOCATOR
     feature
      spy: SPY is
         once
          !! Result.init;
         end;
end -- class SPYLOCATOR
```

```
class SPYMASTER
  --- class SPY must inherit from this class
   creation
     init
   feature
     init is
        deferred
       end;
     c_monitor(caller, callee:ANY; nameOfCallee,
             classOfCallee, procedureCalled,
             listOfArguments :STRING) is
        deferred
       end;

     g_monitor(caller, callee:ANY; nameOfCallee,
             classOfCallee, procedureCalled,
             listOfArguments :STRING) is
         deferred
       end;
end -- class SPYMASTER
```

The class SPY of which spy is an instance, is to be customized for the project at hand, by inheriting from SPYMASTER, which defines the c_monitor and g_monitor features in deferred form. In order that the class SPY provides the desired functionality expected from spy, several rules are necessary. These rules are presented in Figure 9.1.

Because spy is a once feature, a single spy will be available at run time to the instances of the classes that inherit from SPYLOCATOR. By making all user-defined classes in a system inherit from SPYLOCATOR, we can safely say that the spy object is available to the system at run time, since these are the classes that can be instrumented to call spy. The following rule, which rejects classes that does not inherit from SPYLOCATOR

$\mathcal{R}$84. `cannot_include(C,F) :-`
`className(spy)@C,`
`not (inherits(X)@C, className(spymaster)@X).`

*Class* SPY *must inherit from* SPYMASTER.

$\mathcal{R}$85. `cannot_rename(C,_,T) :- className(spymaster)@T| clasName(spy)@T.`

*Features defined in* SPY *or* SPYMASTER *have unique names.*

$\mathcal{R}$86. `cannot_redefine(C,_,T)`
`:- className(spymaster)@T| clasName(spy)@T.`

*Features defined in* SPY *or* SPYMASTER *cannot be redefined.*

$\mathcal{R}$87. `cannot_undefine(C,_,T)`
`:- className(spymaster)@T| className(spy)@T.`

*Features defined in* SPY *or* SPYMASTER *cannot be suppressed.*

$\mathcal{R}$88. `cannot_changeExp(C,_,T)`
`:- className(spymaster)@T| className(spy)@T.`

*Features defined in* SPY *or* SPYMASTER *retain their original export status.*

Figure 9.1: Rules Controlling Customization of spy

included in a system, can be used in this regard:

```
R89. cannot_include(C,F) :-
           not (inherits(X)@C, className(spylocator)@X).
```

In the following section we present how all these can be used together.

### 9.2.2 How To Monitor Eiffel Interactions

In order to monitor Eiffel interactions in a project using the facilities we just described, there are only a few things to do:

- The rules described in the previous section must be included in the system sublaw.
- The class `SPY` must be created, inheriting from `SPYMASTER`, effecting the deferred features `init`, `c_monitor` and `g_monitor` according to the monitoring needs of the project, and declaring `init` as the creation routine. Other classes that `SPY` needs to work with in order to analyze the reported information should also be written.
- The Darwin-E primitive `$do(monitor(_,_,_,_))` should be included in appropriate `can_call` and `can_generate` rules, to specify which interaction will be monitored. We have shown example rules using `$do(monitor(_,_,_,_))` primitive earlier.
- The classes `SPY`, `SPYLOCATOR`, `SPYMASTER` and suppliers of `SPY` must be included in the configurations. This can be aided by appropriately defined `must_include` rules.

The following is an example of a customized `SPY` class, that does not use any other cohorts. The net effect of reports to `spy`, in this simple example, is a message displayed on standard output.

```
class SPY
  inherit SPYMASTER
creation
```

```
    init
  feature
    init is
      do
      end;
    c_monitor(caller:ANY; callee:ANY; n, c, p, l:STRING) is
      do

        io.output.putstring(``Call via '',p, ``to class ``, c,
                            `` is being reported\n'')
      end;
    g_monitor(actor:ANY; newborn:ANY; n, c, p, l:STRING) is
      do

        io.output.putstring(``generation of an instance of
                              class ``, c, `` is being
                              reported\n'')
      end;
  end;
```

There are several salient features that one should be careful about, while designing the class `SPY` and its cohorts. First, the objects reported to `spy` (the first two arguments of `c_monitor` and `g_monitor`) will be of class `ANY`. In the above example, we did not use the objects, but in order to be used effectively, they should be reverse assigned to entities of the appropriate types. For instance, if instances of `ACCOUNT` and `CUSTOMER` are reported to `spy` as `callee` and `caller` respectively, then in order to use them as account and customer we will need the following code fragment inside `c_monitor`:

```
-- declare the following local entities
            a: ACCOUNT;
            c: CUSTOMER;
....
....
-- retrieve account and customer from callee and caller
            a?= callee;
            c?= caller;
....
....
```

Second, monitoring activities should not change the computational state of $\mathcal{S}$. Care should be taken in implementing the routines of `SPY` and its suppliers to ensure such non-intrusiveness. To substantiate this, we may formulate the law in such a way that they can call the reported objects by only certain designated features that are known to be read only. In Chapter 11, we will introduce the notion of *Side-Effect-Free* (SEF)

routines, which could be used in this regard: `spy` and its cohorts can be allowed to use only SEF routines on the reported objects.

Finally, since a software system evolves almost always, we may want to ensure that the desired monitoring is in effect invariant of such evolution. Since what gets monitored is specified by rules in the law, the invariance is guaranteed if these rules themselves are not changed arbitrarily. This means that creation and destruction of the rules that contain the built-in Darwin-E primitive `$do_monitor(_,_,_,_)` should be controlled. This is taken care of by the legislative facilities of Darwin-E.

### 9.2.3 Related Work

The fundamental idea of putting sensors in the code to arrange reporting of events is not new. Refer to [64], for a survey of on-line monitoring. Code instrumentation is a common technique to embed such sensors.

In many cases the instrumentation is done on the object-code. Consider for instance, the case of setting spy points or traces: one needs to compile the program in a special way so that this could be arranged by the debugger. However, source code instrumentation like ours is not uncommon. For instance, in [63], source-code transformation by a pre-processor is used to arrange for monitoring. This is similar to our approach in some sense, but the pre-processor transforms ANA annotations, that are voluntarily included in ADA programs.

Use of a special purpose entity like our `spy`, can be found in [21], where the main objective of monitoring a system is to detect failures and roll-back to a previously checkpointed state in the event of one. Their `watchd` expects *I am alive* messages from the monitored processes. A library routine provides facilities for intermittent transmission of *I am alive* messages, which must be incorporated in the processes being monitored.

In fact, in most of the monitoring systems, a degree of voluntary cooperation of the programmers are needed to arrange the monitoring. The novelty of our work is that the code instrumentation is carried out by the environment *without any knowledge* of the programmers that develop the system being monitored. This kind of *independent*

monitoring is used in financial institutions as a means to build public confidence in their operations and it has been argued in [40] that analogous means should be used as a trust building measure for todays computer systems as well.

# Chapter 10

# An Example Project Under Darwin-E

We would like to demonstrate how the various facilities of Darwin-E can be used in a project. To do so, we will first describe the project informally, and then, formulate it under LGA. We will then describe the initial state of the project, including the law, and finally, we will describe how such a project will work.

## 10.1 Informal Description

The goal of this example project is to build an on-line auditable financial system, first introduced in [40] and described below. The project consists of two divisions namely, the *base division* and the *audit division*. Each division consists of programmers, and the artifacts (classes, configurations etc.) they build. We will often use terms like base programmers, base classes or base configurations to mean programmers, classes and configurations belonging to the base division, and auditors, auditing classes and auditing configurations for the programmers, classes and configurations in the audit division. Each division has a manager who is in charge of his division.

The base classes are the classes that implement the basic functionality of the financial system, typical examples are `CUSTOMER`, `ACCOUNT` etc. The auditing classes are the classes that implement various auditing tools, that are used to audit the transactions made by the financial system. In this example, such tools will run as separate Eiffel processes and maintain their own persistent database of customers and accounts.

We will be using the monitoring facilities of Darwin-E to audit the financial system on-line. Refer to Figure 10.1 for an illustration of how such an auditable financial system may operate. The solid arrow-headed lines denote (call) interaction between objects. The dashed line indicates reporting to `spy`. The financial system itself can

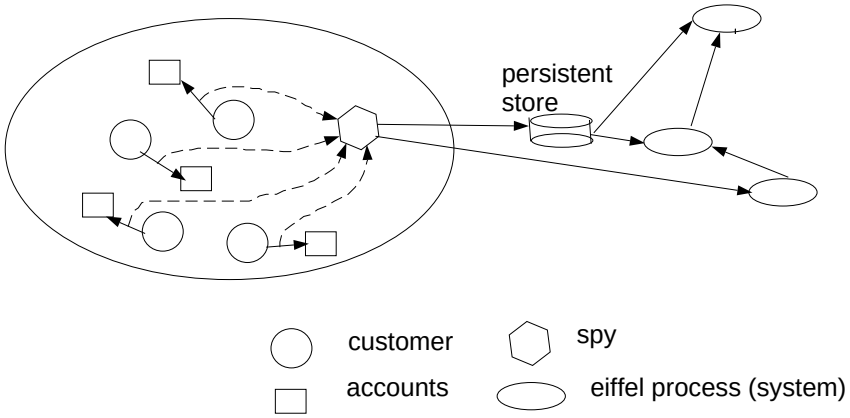


Figure 10.1: An On-Line Auditable Financial System

be conceived as a collection of several customer and account objects and the special purpose object `spy`. Customers will operate on the accounts by sending messages such as `deposit` or `withdraw`. The auditing tools are separate Eiffel processes. The auditors will be responsible for defining which customer-account calls are to be reported to `spy`. They will also be responsible for defining the class `SPY`, which works in conjunction with the other auditing classes to provide various auditing services.

The following are some example services that the auditors may employ. There will be a tool called `display`, which will filter the transactions reported to `spy` according to some logic, and display information about the selected ones. The auditors can use this tool to watch over activities of suspected customers on sensitive accounts, based on some definition of *sensitiveness* and *suspiciousness*. As an example, we have decided that the auditors will manually declare a customer as a suspect and the accounts whose balance exceeds twenty thousand dollars will be automatically treated as a sensitive account. Besides, there will be a tool `irs` which will store records of transactions performed on accounts – an on-line equivalent of the paper trail of customer activities. Finally, there will be another tool `query`, which will run queries on the persistent database of accounts and customers that the auditing tools maintain independent of the financial system. The `spy` object is responsible for updating this database, based on the reports it receives.

## 10.2 Formulation Of The Project Under LGA

The first step to start such a project under Darwin-E is to formulate it under LGA. Such formulation consists of identifying the desired constraints that we intend to impose by law upon the different aspects of the project, ranging from desired regularities of the product to its construction process.

### 10.2.1 Desired Regularities

The very nature of the software being developed in this project, implies that it should possess the following regularities:

- Base classes cannot inherit from auditing classes and vice-versa.
- Base classes cannot call auditing classes.
- Auditing classes should be able to call base classes, but such calls should not change the state of the financial system.

Inter-division inheritance is not allowed in order to make sure that the data structures and methods developed for one division are not assimilated by the other, which is an essential requirement of auditing. The base classes are not allowed to call auditing classes, because that way the system being audited can manipulate the auditing process. The auditors, on the other hand, may need to know informations such as account balance, but they should not be allowed to modify it, because otherwise, information within the financial system can be changed by the auditing classes. In this example, only certain designated features of the base classes can be called by auditing classes.

To conclude this section, we would like to make the following observation. The three classes postulated by the monitoring mechanism of Darwin-E, `SPYLOCATOR`, `SPYMASTER` and `SPY`, require special treatment in the light of the above regularities. The first two classes, supplied by Darwin-E, and the third, being defined by the auditors, although considered otherwise as auditing classes, are not subject to the above constraints. This is required, because otherwise, base classes cannot inherit from `SPYLOCATOR` and calls

to SPY that result from code instrumentation will be illegal – both of which are essential for on-line monitoring.

### 10.2.2 Monitoring Needs Of The Project

The monitoring needs of this project are depicted in Figure 10.2 (the three special classes SPYMASTER, SPYLOCATOR and SPY are shown in black, and in the middle of the two divisions), and can be summarized as follows:

- Activities in the base division will be observed by the auditors.
- Calls from customers to accounts will be reported to spy.

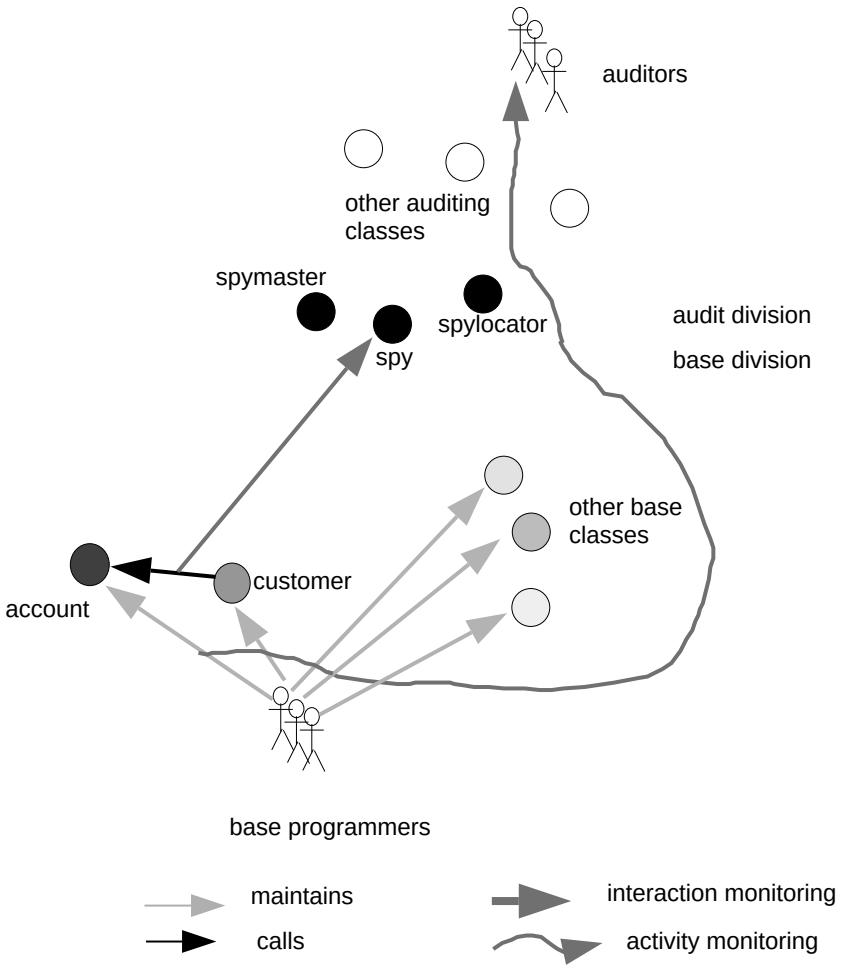


Figure 10.2: Monitoring Requirements of the On-Line Auditable Financial System

The auditors need to watch over the base division in order to keep themselves abreast of the latest situation: they may want to be notified whenever a new class is added in the base division or a change in the class `CUSTOMER` or `ACCOUNT` occurs. This may act as a trigger of a review of the auditing classes to see if any change is required in these classes as well.

A customer operates on an account by means of feature call, therefore auditing tools should be able to monitor these calls. Exactly which such calls will thus be monitored, are to be decided by the auditors. In addition auditing tools may need to retrieve data from accounts or customers, to perform their auditing duties. However, we must ensure that such operations do not alter the data contained in these objects.

### 10.2.3 Configuration Issues

We make the following simplifying assumption in this regard:

*Configurations will be build in this project by hand-picking the user-defined classes as needed, however each base configuration must include the three special classes* `SPYMASTER`, `SPYLOCATOR`, `SPY` *and the other cohorts of* `SPY` *as the requirements of the project may be.*

This simplifies the law of the project, because we will not be needing rules for consensus-based configuration binding and version management.

### 10.2.4 Control Over Builder Activities

When we think about the builder activities in this project, the following two constraints seem very natural:

- The structure of the project (which in fact is the structure of the object-base of the project) should never be compromised.

- The two divisions should evolve disjointly, with the following exception: auditors should be able to observe objects in the base division.

The need for the first constraint is self-evident: objects in the audit division (auditors or auditing classes) cannot pose and act as base objects (base programmers or base classes, respectively) and vice versa. The second constraint, on the other hand, establishes a firm guideline about the capabilities of the builders in each division. The exception allows the auditors to look at base classes in order to design auditing classes, but no such capability is required for base programmers.

The above general guideline, however, does not completely specify exactly what a builder can or cannot do. To make things simple, we assume the following for this project:

- A builder can do whatever he wants to do to the objects in his division, as long as he does not violate the above guidelines.

### 10.2.5 Legislative Needs

It is clear that the structure of the object-base, the desired regularity and monitoring needs, should be maintained throughout the life of the project. This means we have to be careful in defining the law of the project so that the desired constraints indeed become invariants of evolution. In our formulation, we noted that builders will be creating and deleting rules during the life of the project, therefore a fixed law is inappropriate for this project and the initial law must permit (some) legislative activity. We identified that the auditors should be allowed to create/destroy rules that specify

- which base division activities will be observed by the auditors, and
- which customer-account calls will be reported to `spy`.

We will not allow any other legislative activity in the project, therefore the the constraints we formulate in the initial law cannot be compromised later.

## 10.3 The Initial State of The Project

After the server for a new project boots up, it starts with an empty project, with all the built-in objects in its object-base and with the `#creator` as the active builder. It is the

`#creator` who is responsible for putting the project into its initial state, by creating the initial law of the project and some initial objects (builders, classes, etc) that are needed to set the ball rolling.

The initial law $L_1$ of this project contains the rules and metarules that realize the LGA formulation described earlier. In the following subsections, we will present $L_1$ in several installments. As we shall see, the law makes use of the property `manager` in a builder to designate him as the manager of a division, and the property `division(X)`, where `X` is either `base` or `audit`, to designate the association of a Darwin-E object with a division.

Let us assume that after creating the initial law, `#creator` will create the managers of the two divisions as well, which means that they are responsible for actually starting the activities in the project.

### 10.3.1 Law of The Project: Regulating The Builders

The first fragment of $L_1$ (figure 10.3) contains the rules that regulate the activities of the builders involved in the project. These rules strictly enforce the structure of the two divisions, defines the capabilities of the builders in each division, and grants the auditors read-only access to the base division objects.

Comments after individual rules provide sufficient explanation, however, we wish to say some more about the rules $\mathcal{R}93$ and $\mathcal{R}92$. These rules establish the desired capabilities of the auditors. Rule $\mathcal{R}93$ allows them to issue any observer command on the objects in base division and rule $\mathcal{R}92$ invokes the auxiliary rule $\mathcal{R}97$ for the activities of base programmers on base objects that satisfy `monitor_operation(_,_,_)` goal. The rule $\mathcal{R}97$ ensures that such activities are reported to the manager of the audit division and are archived in `#tracer`. The `monitor_operation(_,_,_)` goal is defined by `monitor_operation` rules – which as we shall see, will be created by the manager of the audit division.

$\mathcal{R}90$. `canDo(S,new(X,_),T) :- (T= class|T= configuration),`
`division(D)@S,`
`$do(set(division(D))@X).`

*Programmers can create classes and configurations in their respective divisions.*

$\mathcal{R}91$. `canDo(S,M,T) :- division(audit)@S,division(audit)@T,`
`not forbiddenOp(M).`

*Auditors can operate freely on objects in audit division, except for the forbidden operations defined by the* `forbiddenOp` *rule (see rule $\mathcal{R}96$ below).*

$\mathcal{R}92$. `canDo(S,M,T) :- division(base)@S,division(base)@T,`
`not forbiddenOp(M),`
`(monitor_operation(S,M,T) -> trace_and_ucast(S,M,T) |`
`true).`

*Base programmers can operate on objects in the base division freely, except for the forbidden operations, and such activity will be monitored if a* `monitor_operation` *rule mandates so.*

$\mathcal{R}93$. `canDo(S,M,T) :- division(audit)@S,division(base)@T,`
`observerOp(M).`

*Auditors are allowed read access on objects in base division.*

$\mathcal{R}94$. `canDo(S,M,T) :- division(audit)@S,traceOp(M,_,_,_).`

*Auditors can see and destroy the records of the monitored activities.*

$\mathcal{R}95$. `canDo(S,new(X),builder) :- role(manager)@S,`
`division(D)@S,`
`$do(set(division(D))@X).`

*A manager can create new builders for his division.*

$\mathcal{R}96$. `forbiddenOp(X) :- X= createRule(_,_)| X= createMetaRule(_,_,_)|`
`X= set(division(_))| X= recant(division(_))|`
`X= set(role(_))|X= recant(role(_)).`

*Auxiliary rule that defines some activities as forbidden, this will be used in rules $\mathcal{R}92$ and $\mathcal{R}91$.*

$\mathcal{R}97$. `trace_and_ucast(S,M,T) :- $do(trace(S,M,T)),`
`division(audit)@R,role(manager)@R,`
`$do(ucast(message(S,M,T))@R).`

*Auxiliary rule that performs sends a message to the audit manager and archives the activity in* `#tracer`.

Figure 10.3: $L_1$ (a) Regulating Software Development

### 10.3.2 Law of the Project: Regulating Eiffel Interactions

The next fragment of $L_1$ (refer to Figure 10.4) contains rules that regulate Eiffel Interactions (essentially the system sub-law of the project). Although sufficient explanation is provided by the comments after individual rules, we would still like to make the following observations. First, we assumed that the identity of a module object is same as the name of the class that it represents – a simplifying assumption we used earlier as well. Second, to avoid repetition, the prohibition rules described in Figure 9.1 of Chapter 9, concerning `SPY, SPYMASTER` and `SPYLOCATOR`, are not shown here, but these rules are assumed to be present in the initial law. Third, note that the Figure 10.4 contains rules about `include, call` and `inherit` interactions only. The LGA formalization does not call for any restriction on the regulated Eiffel interactions that are not mentioned in Figure 10.3, and we assume that these interactions are permitted freely. To make the exposition simple, we will not show the rules that implement such permission. Note however, that, this does not invalidate the prohibitions concerning `SPY, SPYMASTER` and `SPYLOCATOR`, that are present in the law, as mentioned earlier. Fourth, of special interest is Rule $\mathcal{R}102$, that provides a facility to specify which customer-account calls will be reported to `spy` by defining `monitor_call` rules– as we shall see, such rules can only be defined by the auditors. Finally, in Rule $\mathcal{R}103$ we allowed auditing classes to call base classes assuming that the features listed will never attempt to change the state of the callee object. This simplistic assumption can be avoided by allowing auditing classes to call base classes via SEF routines, to be introduced in Chapter 11.

#### Law of The Project: Control Over Legislation

We have stated earlier that the manager of the audit division is responsible for creating the `monitor_operation` and the `monitor_call` rules. The `monitor_opMr` and the `monitor_callMr` metarules, from which the `monitor_operation` and `monitor_call` rules will be created respectively, should be defined in the initial law, but we will not show them here. The rule $\mathcal{R}104$ in figure 10.5 will allow the manager of $P_a$ to create

$\mathcal{R}$98.
```
must_include(C,[irs,spy,spylocator,spymaster]) :-
        division(base)@C.
```
*In addition to the three classes* `spy, spymaster,spylocator`*, all configurations in the base division must include the (auditing) class* `irs`*.*

$\mathcal{R}$99.
```
can_include(customer,F) :-
        division(base)@F,
        inherits(spylocator)@C.
```
*The class* `customer` *can be included in a base configuration only if it inherits from* `spylocator`*, this mean that the special purpose object* `spy` *will be available to its instances at run time.*

$\mathcal{R}$100.
```
can_inherit(C1,C2) :- division(X)@C1,
        (division(X)@C2| library@C2).
```
*A class inherit from another class in its own division or in the Eiffel library.*

$\mathcal{R}$101.
```
can_call(_,C1,_,C2) :- division(audit)@C1,division(audit)@C2.
```
*Auditing classes can call each other freely.*

$\mathcal{R}$102.
```
can_call(F1,C1,F2,C2) :- division(base)@C1, division(base)@C2,
        (monitor_call(F1,C1,F2,C2) -> $do(monitor(F1,C1,F2,C2)) |
        true ).
```
*Classes in base division can call other classes in the same division, but such an interaction will be reported to* `spy` *(by means of the code instrumentation effected by the* `$do(monitor(_,_,_,_))` *primitive) if indicated by a* `monitor_call` *rule.*

$\mathcal{R}$103.
```
can_call(_,C1,F,C2) :- division(audit)@C1,
        division(base)@C2,
        in(F,[name,acNum,balance,lastDeposit,..]).
```
*Auditing classes can call base classes via features that are listed above.*

Figure 10.4: $L_1$ (b) Regulating Eiffel Interaction

such rules. Note that only the `monitor_operation` and `monitor_call` rules are creatable (or destructible) in this project. This and the strict enforcement of the law ensures that the monitoring needs established by the law $L_1$ remains invariant of evolution. We will see more about this aspect of the project in the next section.

$\mathcal{R}$104.

```
canDo(S,createRule(X,_),T) :- role(manager)@S,
      division(audit)@S,
      (T= monitor_opMr| T= monitor_call).
```

*Manager of $P_a$ can create rules from* `monitor_opMr` *and* `monitor_callMr` *metarules.*

$\mathcal{R}$105.

```
canDo(S,removeRule,T) :- role(manager)@S,
      division(audit)@S,
      #legislator(S)@T.
```

*The built-in property* `#legislator(b)` *in a rule* `r` *denotes that builder* `b` *created* `r`. *This rule allows the manager of $P_a$ to destroy the rules he created.*

Figure 10.5: $L_1$ (c) Controlling Rule Creation and Destruction

### 10.3.3 How Does It All Work

The law of the project allows the managers of the two divisions to appoint new builders in their divisions. So, after the law of the project is created, the two managers can log in and introduce the builders they need. Each division then can work independently under the control of the rules presented in figure 10.3. For instance, base programmers will create classes representing customers and accounts and applications involving them. Similarly, the auditors will customize the class `SPY` and create its cohorts. In this project, `SPY` will be customized in such a way that when a customer-account call reported, the following things will happen:

1. a record is filed with `irs` (an instance of class `IRS`, created by the auditors),

2. the persistent database of accounts and customers maintained by auditors will be updated, and

3. a message about the customer and the account is send to `display`, which is another tool created by the auditors.

If `display` is not started by the auditors, the last action above does not have any effect. However, if `display` is running, then the reported account and the customer will be analyzed. If it was a suspicious customer operating on a sensitive account, this news will be displayed at the console of the auditor running the `display` tool. In addition, the auditors are also responsible for creating other auditing tools: tools they will need for declaring a customer to be suspicious, or making queries on the persistent auditing database.

At some point, the manager of the audit division would wish to monitor some of the activities of the members of the base programmers and may create the `monitor_operation` rules appropriately. Rule $\mathcal{R}106$ in Figure 10.6 is an example of such a rule. As a result of the introduction of this rule, whenever a base programmer creates a base object, or modifies an existing base object, the manager of the audit division will know about it. These events will also be recorded in `#tracer`, and by virtue of the rule $\mathcal{R}94$ in $L_1$, he can use this recorded information later. The auditors can read the code and properties of the base objects, so that they can adapt the code of auditing classes appropriately.

Note that according to the law $L_1$, base programmers can assemble or run a system that belongs to the base division and similarly the auditors can assemble and run systems that belong to the audit division. However, because of rule $\mathcal{R}98$, classes `SPY`, `SPYMASTER`, `SPYLOCATOR` and `IRS` are always included in any base configuration.

Before the base programmers assemble any system, the audit manager should create the `monitor_call` rules that specify which Eiffel interactions will be monitored. For example, let us consider the rules $\mathcal{R}107$ in figure 10.6, which ensures that calls via `deposit` or `withdraw` from customers to accounts in any system assembled in this project is guaranteed to be reported to `spy`. The impact is that any customer and account ever involved in a withdraw or deposit transaction will be subject to examination by the auditing tools via the reportings through `spy`.

$\mathcal{R}$106.

```
monitor_operation(S,M,T) :- createOp(M,_)| modifierOp(M).
```

*Only creation and modification activity is monitored.*

$\mathcal{R}$107.

```
monitor_call(F1,C1,F2,C2) :- className(customer)@C1,
       className(account)@C2,
       (F2= deposit|F2= withdraw).
```

*Class customer calling account with withdraw or deposit will be monitored.*

Figure 10.6: Specific Monitoring Constraints

According to the law $L_1$, no one other than the manager of the audit division can delete the rules $\mathcal{R}$106 or $\mathcal{R}$107. No one else can introduce new `monitor_operation` or `monitor_call` rules either. So the manager of the audit division is responsible for determining exactly what will be monitored.

Since no other rules can be created or destroyed in the project, the way the `monitor_call` and `monitor_operation` rules are invoked, as defined in the initial law $L_1$, cannot be changed. Similarly, there is no way to by-pass the strict enforcement of the other rules in $L_1$.

# Chapter 11

# Examples of Regularity In Use

In this Chapter we present some examples of useful regularities that can be established by concurrent control over several Eiffel interactions. The examples range from regularities that fortify Eiffel programs, to regularities implicit in architectural designs. In each case, we will be presenting a law fragment, that explicates and establishes the desired regularity. To make the rules simpler, we will assume as we did before, that the identity of a module-object is same as the name of the class it represents unless specified otherwise. In most of the examples, we will be using prohibition based control, showing in the law fragments only the rules that express the prohibitions.

## 11.1 Fortifying Eiffel

Examples in these section include some useful notions that can be adopted for Eiffel programs, but there is no support from the language to implement them. By formulating these as regularities, we guarantee that the Eiffel programs conform to the constraints underlying these notions.

### 11.1.1 Immutability

Consider the following notion of immutable class:

**Definition 13 (immutability)** *A class* c *is said to be* immutable *if (a) all its instances are immutable, and (b) if attributes defined in class* c *are immutable even as a component of an instance of a descendant of* c.

Note that this concept of immutability is a *regularity* in our sense of this term, since it cannot be localized in any given class, or in any fixed set of classes. Indeed, while it is possible to satisfy property (a) of this definition by appropriate construction of class c itself, the satisfaction of property (b) depends on all descendants of c. However, such a property can be ensured by a law, as we shall see below.

$\mathcal{R}$108.

```
cannot_inherit(C1,C2) :-
        immutable@C1,
        not immutable@C2.
```

*An immutable class cannot inherit from a non-immutable class.*

$\mathcal{R}$109.

```
cannot_assign(_,_,_,C) :- immutable@C.
```

*Prohibition of assignment to attributes of a class marked as* `immutable`.

$\mathcal{R}$110.

```
cannot_call(_,_,F,C) :-
        creation(F)@C,
        immutable@C.
```

*Prohibition of regular calls of creation routines of classes marked as* `immutable`.

Figure 11.1: Establishing a Concept of Immutable Class

The law-fragment of Figure 11.1, makes any class marked as `immutable` into an immutable class in the sense of Definition 11.1.1. This is done as follows: Rule $\mathcal{R}$108 prohibit classes marked as immutable from inheriting from classes not so marked, for obvious reason. Rule $\mathcal{R}$109 prohibits assignment to the attributes defined in such a class. These two rules should have been sufficient for immutability, except for the following problem: According to Definition 9 the prohibition on assignments exempts the creations routines of a class. This does not contradict immutability as long as the creation routines are not called as normal routine on an already initialized object, which is permitted in Eiffel. Such use of creation routines is prohibited by Rule $\mathcal{R}$110.

### 11.1.2 Private Features

Let us defined the concept of a *private feature* of a class as follows:

**Definition 14 (private feature)** *A feature* `f` *defined in class* `c` *is called a* private feature *of* `c` *if it is accessible only in routines defined in* `c` *itself.*

$\mathcal{R}$111.

```
cannot_assign(_,C1,F,C) :-
        private(F)@C,
        C1=/=C.
```

*Prohibition of assignments to private attributes of class* `C` *by any other class.*

$\mathcal{R}$112.

```
cannot_call(_,C1,F,C) :-
        private(F)@C,
        C1=/C.
```

*Prohibition of calls to private attributes of class* `C` *from any other class.*

Figure 11.2: Establishing a Concept of Private Feature

This useful notion is supported by both Simula 67 [6] and C++ [35], but unfortunately not by Eiffel, in which features of a class are automatically visible in all the descendant of this class. This limitation of Eiffel can be easily rectified under Darwin-E. In particular, the pair of rules in Figure 11.2, would make any attributes `f` of class `c` private if `c` has the property `private(f)` in the object-base $B$ of project $P$ governed by this law fragment.

### 11.1.3 Side-Effect-Free Routines

It is sometimes useful to have the assurance that a certain kind of routines are *side-effect-free* (SEF); that is, that they have no effect on the state of the system beyond the result being returned. (It is, of course only useful for functions to be SEF.) A case in point are the functions used for on-line monitoring: such functions are not supposed to change the state of the system they are monitoring. In Chapter 9 and 10, we have described cases where the guarantee that a function is SEF is crucial.

But how do we know which routines are SEF? Of course, one can program any given routine carefully to be SEF and then allow it to be used as such (say for on-line monitoring). But how do we know that the given routine would retain its SEF nature throughout the evolutionary lifetime of the system? One solution to this problem is

given by the law-fragment in Figure 11.3. This set of rules makes sure that if a class `c` has the property `sef(r)` then the Eiffel routine `r` defined in `c` is a SEF routine[1].

$\mathcal{R}$113.

```
cannot_assign(F,C,_,_) :- sef(F)@C.
```

*A SEF routine should not perform any assignments (except assignments to local variables, which are not controlled by this rule).*

$\mathcal{R}$114.

```
cannot_generate(F,C,_,_) :- sef(F)@C.
```

*A SEF routine is not allowed to create new objects*

$\mathcal{R}$115.

```
cannot_call(F1,C1,F2,C2) :- sef(F1)@C1,
      not sef(F2)@C2,
      not defines(attribute(F2),_)@C2,
      not certified_as_sef(F2)@C2.
```

*A SEF routine* `F1` *cannot call* `F2` *unless it is also a SEF routine, or it is an attribute (and thus inherently SEF), or it is* certified *as SEF routine.*

Figure 11.3: Establishing the Concept of Side Effect Free (SEF) routine

Rule $\mathcal{R}$113 of this law-fragment prohibits SEF-routines from making any assignments into attributes of an object, which includes prohibition of instantiations into attributes. Rule $\mathcal{R}$114 prohibits all instantiations by SEF routines, even instantiations into local variables of a routine (note that assignment to local variable is not prohibited by this law.) Finally, Rule $\mathcal{R}$115 does not let a SEF routine `f1` call another routines `f2` unless (a) `f2` is also a SEF routine, or (b) `f2` is an attribute (and thus inherently SEF), or (c) `f2` is *certified* as SEF routine. The third possibility refer to a property `certified_as_sef(f2)` of a class `c2` where `f2` is defined as a C-coded routines. The point here is that our law does not analyze C-coded routines, which thus require their SEF status to be certified by one of the builders of the system. Such certification can, of course, be regulated by the law of the project.

[1] We assume here that C-coded routines cannot be marked in this way, which can easily be ensured by the law under Darwin-E.

## 11.2 Regularities Implicit In Architectural Design

In this Section we will consider two well known architecture designs namely, *layered design* and *kernelized design*. In each case we identify the underlying constraints implicit in the architectural design model and show how such constraints can be established as regularities.

### 11.2.1 Layered Design

The constraints underlying a layered design depends on the notion of a *layer*, which in our context, consists of a set of Eiffel classes. A layer is entrusted with the responsibility of implementing a set of services that is made available to the layer above it. Crucial to this objective are the following constraints:

> **Constraint1:** *Inheritance across layers is not allowed.*
>
> **Constraint1:** *A class* `c1` *in a layer* `l1` *can declare entities of another class* `c2` *belonging to layer* `l2` *only if* `l1` *and* `l2` *are in the same layer or the layer* `l2` *is immediately below* `l1`.

Violation of the first constraint allows any layer to assimilate the features (services) that was designed to be a part of a different layer. Similarly, violation of the second constraint allows the classes in a layer to use services that are implemented in any arbitrary layer. In both cases the nice structure of layers will be lost.

Conventional programming languages do not provide any support for designating layers and imposing global constraints such as the above on the basis of layers. Consequently, the constraints are expected to be programmed into the layered system, which makes the very architecture unreliable, invisible, hard to maintain and reason about.

In Darwin-E however, we can assign the property `layer(l)` to classes that belong to layer `l` and enforce the rules presented in Figure 11.4. These rule express the desired regularity a layered architecture discerns. We assume that the layers are numbered bottom up, the lowest layer has the lowest layer number and the layer number is incremented by one as we go one layer up.

$\mathcal{R}$116.

```
cannot_inherit(C1,C2) :- layer(L1)@C1,
       layer(L2)@C2,
       L1 ≡ L2.
```

*inheritance from a class in a different layer is illegal.*

$\mathcal{R}$117.

```
cannot_use(C1,C2) :- layer(L1)@C1,
       layer(L2)@C2,
       (L1> L2+1 | L1< L2).
```

*a class in a layer cannot declare entities of a class that belongs to a higher layer or a layer that is more than 1 layer below.*

Figure 11.4: Law of Layered Design

Different variations of the layered design can be thought of and be readily incorporated into the law. We will show one example below.

In this variation, the classes in a layer are allowed to use some selected services of the layer immediately above, in addition to the services of their own layer and the layer immediately below it. The ability to call upwards is often necessary when a layer needs to call back or interrupt an upper layer. To accommodate this, we now need to allow classes to declare entities of a class that belongs to the layer immediately above in addition to the classes that are in the same or immediately below layer. The rule $\mathcal{R}$117 in Figure 11.4 will be replaced by the rule below:

$\mathcal{R}$118.

```
cannot_use(C1,C2) :- layer(L1)@C1,
       layer(L2)@C2,
       (L1>L2+1|L2>L1+1).
```

In addition, calls made to the above layer need to be controlled: only specific features, which are marked by the `upcall(_)` property, are to be allowed in an upcall. This is

achieved by introducing the following prohibition on call interactions:

$\mathcal{R}$119.

```
cannot_call(_,C1,F,C2) :- layer(L1)@C1,
       layer(L2)@C2,
       L2 = L1+1,
       not upcall(F)@C2.
```

The regularity imposed by the above rules depends heavily on properties like `layer(_)` or `upcall(_)`. Consequently, if these properties are manipulated arbitrarily, the regularities become meaningless. Fortunately, under Darwin-E, it is possible to impose a tight control over such manipulation. In particular, it is very easy to restrict the ability to manipulate these properties to a responsible person such as the project manager:

$\mathcal{R}$120.

```
canDo(S,set(P),T) :-
       (P= layer(_)|P= upcall(_)),
       manager@S.
```

### 11.2.2 Kernelized Design

The notion of kernelized design is used quite often in the context of embedded system, where a kernel provides a tamed abstraction of the underlying hardware to the rest of the system. The operating system of a computer is perhaps the best known embedded system. The objective of a kernelized design is to localize the critical and complicated tasks in the kernel, so that various applications can easily be built reliably and relatively easily using the kernel services.

The constraints implied in kernelized design are presented below:

**Constraint1:** *The kernel should have exclusive access to the underlying hardware.*

**Constraint2:** *The kernel should be independent of the rest of the system.*

**Constraint3:** *The kernel should be usable by the rest of the system only in a well-defined manner.*

The reason behind the first constraint is obvious, we certainly do not want every piece of code to manipulate the hardware. If an embedded system is built using Eiffel, the only way to access the hardware is via external C routines that can make system calls. Therefore this constraint amounts to restricting the availability of C coded routines only to the kernel classes. This does not cause any loss of generality, since as argued in Chapter 6, use of C coded routines in Eiffel classes is very unsafe and if possible should be avoided. This constraint suggests that the cases where we must use C coded routines, should be localized in the kernel classes. Let us assume that we will have a cluster of such classes, designated by the property `cluster(kernel)`. The second constraint is necessary to make the abstraction of the hardware provable on the basis of the code in the kernel alone. If the kernel uses (inherits or calls) non-kernel classes, the basis will not be closed. The third constraint is necessary to allow the kernel to have some of its features accessible to classes in the kernel, but hidden from the rest of the system. The selective export of Eiffel, is not sufficient in this case because it demands explicit naming of classes in the export clause, where as we want such specification by intention.

The law that captures the regularity expressed in the constraints is presented in Figure 11.2.2.

The first constraint is established by rule $\mathcal{R}121$, which allows only kernel-classes to have C-coded routines. The second constraint is established mostly by rule $\mathcal{R}122$, which prohibits kernel classes from being clients of (i.e. to declare entities of) non-kernel classes, and by rule $\mathcal{R}123$ which prohibits kernel classes from inheriting from non-kernel classes. Rules $\mathcal{R}124$ and $\mathcal{R}125$ can also be viewed as contributing to this principle. These rules ensure that features defined in the kernel have a kind of *universal semantics*, by prohibiting their redefined and renaming anywhere except in the kernel itself. The third constraint is established by rules $\mathcal{R}126$ and $\mathcal{R}127$. Rule $\mathcal{R}127$, in particular, allows non-kernel classes to call the kernel only by means of features marked explicitly as `interface_feature`. But this rule is not quite sufficient because of the ability of a non-kernel classes to inherit from a kernel class, and then to assign to its attributes. This capability is prohibited by rule $\mathcal{R}126$.

$\mathcal{R}$121.

```
cannot_useC(D,_) :- not cluster(kernel)@D.
```

*C-code cannot be used outside of the kernel*

$\mathcal{R}$122.

```
cannot_use(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```

*kernel classes cannot use (be client of) non-kernel classes.*

$\mathcal{R}$123.

```
cannot_inherit(C1,C2) :-
        cluster(kernel)@C1,
        not cluster(kernel)@C2.
```

*kernel classes cannot inherit from non-kernel classes*

$\mathcal{R}$124.

```
cannot_redefine(C1,_,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2.
```

*Features of kernel-classes cannot be redefined by non-kernel descendants.*

$\mathcal{R}$125.

```
cannot_rename(C1,F,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2,
        exports(C2,F).
```

*Exported features of kernel-classes cannot be renamed by non-kernel descendants.*

$\mathcal{R}$126.

```
cannot_assign(_,C1,_,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2.
```

*Attributes of kernel classes cannot be assigned to by non-kernel classes.*

$\mathcal{R}$127.

```
cannot_call(_,C1,F2,C2) :-
        not cluster(kernel)@C1,
        cluster(kernel)@C2,
        not interface_feature(F2)@C2.
```

*Features of kernel classes cannot be called from non-kernel classes* unless *they are marked as interface_features.*

Figure 11.5: Law of Kernelized Design

Note again that the rules are meaningful if the properties like `cluster(kernel)` or `interface_feature(_)` are set responsibly. Otherwise any class can be put into the kernel cluster or any feature of a kernel class can be declared an interface-feature. It should be obvious by now, that under Darwin-E it is not a problem: a tight control over such properties can be established by the law of the project.

## 11.3 Supplementing Design Patterns

Design patterns [16, 59] are idiomatic object-oriented structures that promote good design. A design pattern is essentially an architectural template that can be realized in different contexts. It consists of a small set of classes that are designated to play well-defined roles.

There are certain constraints that must be followed in realizing a pattern. Such constraints are usually stated informally in the pattern documentation by means of phrases like *no class except* `C` *can create ...* or *all these classes must conform to ..*, which cannot be dealt with by conventional programming languages. This does not however, mean that effective realization of patterns are impossible: a pattern can be successfully realized by careful programming, but then the onus will be on the programmers who must understand, extract and follow these constraints. Such a manual approach to programming global constraints into a system is inherently unreliable, and hard to implement and reason about. We therefore argue that to complement the effectiveness of patterns, these constraints should be formal, explicit and enforced. We will demonstrate that this can be done by formulating the constraints implied by several example patterns as regularities. However, we do not claim that it is possible or even worthwhile to enforce all constraints in the realm of patterns as regularities. There certainly are instances where the constraint is very localized, in which case it can be programmed easily and it is unnecessary to employ the enforcement mechanism. There are also cases where the constraint cannot be expressed as a regularity that we can enforce. We will discuss these in terms of several examples.

### 11.3.1 The Changeable Role Pattern

Our first example is the *changeable role* pattern of Coad [10]. The basic idea of this pattern is that a player plays different roles and assumes a role as if by wearing a hat. For example, consider a polymorphic iterator which can act as a name-iterator or an address-iterator depending on the set it handles. If the elements of the set are chained by name [address] then it becomes a name-iterator [address-iterator]. The participants in this pattern are a `player` class and several `role` classes. The underlying constraints are simple and straight-forward:

> **Constraint 1:** *A player class must be defined in terms of an abstract role class.*
>
> **Constraint 2:** *All role classes for a player must inherit from the abstract role class for the player.*

$\mathcal{R}$128.

```
cannot_use(C1,C2) :- player@C1,
      role(C1)@C2.
```

*A player class (marked by the property* `player`*) cannot declare entities of its role classes.*

$\mathcal{R}$129.

```
cannot_include(C1,_) :- role(P)@C1,
      not (inherits(C2)@C1, abstractRole(P)@C2).
```

*A role class for a player* `p` *cannot be included in a system if it does not inherit from the abstract role class for* `p`.

Figure 11.6: Prohibition Rules in the Law of Changeable Roles

The rules in Figure 11.6 express the regularity captured by the constraints. We used the property `abstractRole(p)` to designate the abstract role class, and the property `role(p)` to designate the role classes for a player class `p`. If there is only one player class p, then this class could be hand-crafted carefully to make sure that the first constraint is satisfied, and having to enforce this constraint as a regularity would be overkill. But if there are multiple player classes, potentially developed by different programmers then

enforcement of the constraints is most desirable. The second constraint, spanning over multiple classes however is better enforced than programmed in either case.

### 11.3.2 The Abstract Factory Design Pattern

The *abstract factory* pattern of Gamma et.al. [14, 16] is our next example. This pattern consists of the following categories of classes[2](refer to the illustration in Figure 11.7 also):

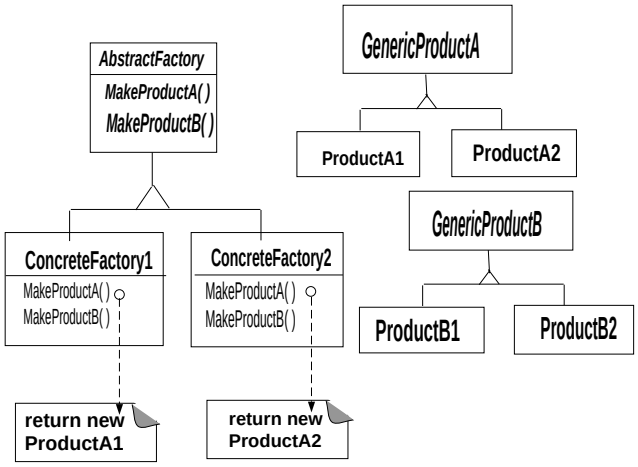


Figure 11.7: Model of the Abstract Factory Design Pattern

- genericProduct: A genericProduct class declares a generic interface for product objects. For example, class WINDOW or class SCROLL_BAR describes the abstract interface of the corresponding graphical entities.

[2]We will be using the terminology used by Gamma et. al. in [16]

- specificProduct: A specificProduct class defines a concrete product, which *must* conform to the corresponding genericProduct. For example, MOTIF_WINDOW is a specificProduct class, defining concrete product objects namely motif windows. MOTIF_WINDOW must conform to the corresponding genericProduct WINDOW.

- abstractFactory: An abstractFactory class declares a generic interface for operations that create genericProducts. A class TOOLKIT with methods `make_a_window` and `make_a_scroll_bar` (with obvious meaning) are examples.

- concreteFactory: A concreteFactory class defines operations that create specific product objects. Classes like MOTIF_TOOLKIT or OPEN_LOOK_TOOLKIT are examples. These classes *must* conform to the corresponding abstractFactory, namely the TOOLKIT; and the method `make_a_window` of MOTIF_TOOLKIT will create an instance of MOTIF_WINDOW. Usually there will be one abstract factory (such as TOOLKIT), to which many concrete factories (such as MOTIF_TOOLKIT or OPEN_LOOK_TOOLKIT) conform.

We now present the constraints implicit in this design pattern:

**Constraint 1:** *Client classes (classes that actually utilize the specific products) should be defined in terms of the generic products. For example an application using graphical objects should deal with WINDOWS rather than MOTIF_WINDOW or OPEN_LOOK_WINDOW.*

**Constraint 2:** *Only concrete factories can create specific product objects. For example, application classes cannot create instances of MOTIF_WINDOW.*

**Constraint 3:** *Specific product classes must conform to the corresponding generic product classes.*

**Constraint 4:** *Concrete factories must conform to the abstract factory, and redefine the inherited methods appropriately to return proper specific products.*

Two comments about the constraints are in order. First, note that categorization of classes such as specific product or generic product is outside the realm of conventional languages such as Eiffel. It follows that the constraints that are based on these notions cannot be supported in such a language. Second, assuming that there exits some means (such as properties associated with class objects in Darwin-E) to designate a class as a specific product or a concrete factory, the last two constraints define the conditions such a class must fulfill. We will now present the Darwin-E rules that express these constraints in Figure 11.8. Rules in Figure 11.8 introduce two auxiliary goals which are defined by the rules shown in Figure 11.9.

One of the constraints in the abstract factory design pattern attempts to localize the creation of specific products within the concrete factories. It is possible in Eiffel, to localize the creation of MOTIF_WINDOW or MOTIF_SCROLL_BAR to MOTIF_TOOLKIT by selective export of creation procedures. Thus the support for constraint 2 (only concrete factories can create specific product objects) is available from Eiffel to some extent. But this is not good enough because of the following two reasons:

- A feature exported to a class C is also exported to heirs of C. Thus, even if the creation routines of specific products are exported to concrete factories, by inheriting from the concrete factory, any class can create specific products in them.

- There is no way to make sure that all specific product classes will correctly export the creation only to the appropriate concrete factory.

Therefore, if the selective export mechanism of the language is used in the realization of the abstract factory design pattern in Eiffel systems, we will need the following additional constraints (in lieu of constraint 2 mentioned earlier):

1. No one can inherit from concrete factories.

2. Specific products can only export creation methods to concrete factories.

$\mathcal{R}$130.

```
cannot_use(C1,C2) :- clientClass@C1,
        specificProduct@C2.
```

*The property* `specificProduct` *designates the classes that define the specific products. The property* `clientClass` *designates the actual users of the specific products. This rule prevents such classes to declare entities of specific product classes.Of course, one might forget to set the* `clientClass` *property and there by by pass this rule. However, as we have shown earlier(Section 5.3.3), we can force this property in the classes when they are created.*

$\mathcal{R}$131.

```
cannot_generate(F1,C1,F2,C2) :-
        specificProduct@C2,
        (not) concreteFactory@C1.
```

*The meaning of the* `concreteFactory` *property is obvious. All classes except for those designated as concrete factory are prevented from creating instances of specific product classes.*

$\mathcal{R}$132.

```
cannot_include(C,_) :-
        specificProduct@C,
        (not) eligible_specific(C).
```

*This rule states that a class with the property* `specificProduct` *cannot be included in any system (denoted by the unnamed Prolog variable* `_`*), if it fails to satisfy the goal* `eligible_specific(_)`*, which is defined by the auxiliary rule $\mathcal{R}$134.*

$\mathcal{R}$133.

```
cannot_include(C,_) :-
        concreteFactory@C,
        (not) eligible_concrete(C).
```

*A class with the property* `concreteFactory` *cannot be included in any Eiffel system if it fails to satisfy the goal* `eligible_concrete(_)`*, which is defined by the auxiliary rule $\mathcal{R}$135.*

Figure 11.8: Prohibition Rules in the Law of Abstract Factory

$\mathcal{R}$134.

```
eligible_specific(C) :-
        inherits(D)@C, genericProduct@D, productType(X)@C,
        productType(X)@D.
```

*Defines the* eligibility *of a specific product: it must inherit from a generic product of the proper product type.*

$\mathcal{R}$135.

```
eligible_concrete(C) :-
        inherits(D)@C,
        abstractFactory@D,
        setOf((X,Y),defines(routine(X),of_type(Y))@D,S),
        check_eligibility(S,C).
```

*Defines the* eligibility *of a concrete factory: it must inherit from an abstract factory and satisfy the constraints expressed by the* `check_eligibility` *rules (see below). We retrieve and pass the names and return types of the routines defined in the abstract factory to the* `check_eligibility` *predicate.*

$\mathcal{R}$136.

```
check_eligibility([],_).
```

*If an empty set is passed, this rule succeeds.*

$\mathcal{R}$137.

```
check_eligibility([(X,Y)|S],C) :-
        (defines(routine(X),of_type(T))@C,heirOf(T,Y))
        → check_eligibility(S,C) | fail.
```

*This rule looks at the first (routine-name, return-type) pair of the input list and if the routine is redefined to return a co-variant type of its original return type it calls itself recursively on the rest of the list. Otherwise it fails.*

Figure 11.9: Auxiliary Rules Used in the Law of Abstract Factory

The Darwin-E rules expressing the above constraints are shown below:

$\mathcal{R}$138.

```
cannot_inherit(C1,C2) :-
       concreteFactory@C2.
```

*The predicate* `cannot_inherit(C1,C2)` *is used to prevent* `C1` *to inherit from* `C2`. *This rule prevents inheritance from concrete factories.*

$\mathcal{R}$139.

```
cannot_exportCreation(C1,_,C2) :-
       specificProduct@@C1,
       (not) concreteFactory@C2.
```

*The* `cannot_exportCreation(C1,F,C2)` *predicate is used to prevent* `C1` *from exporting a creation method* `F` *to* `C2`. *This rule prevents export of any creation method by any specificProduct to classes that are not concrete factories.*

### 11.3.3 The Wrapper Pattern

Our final example is the *wrapper* pattern of Gamma et. al. [14]. The participants in these pattern are called the `component` and the `wrapper`. The `wrapper` object encapsulates and enhances the `component` object by defining an interface that conforms to that of the `component` and maintaining a reference to the `component`. It intercepts the requests for the `component` and forwards it to the `component`.

The implicit constraints in this case are as follows:

**Constraint1:** *A wrapper class must conform (in terms of Eiffel inherit) to the class of its component.*

**Constraint2:** *A wrapper must have a reference to its component.*

**Constraint3:** *A wrapper must redefine the inherited routines (to dispatch to the component).*

Note that if there is only one `wrapper`, encapsulating a single `component` then such constraints can easily be programmed in the class of the `wrapper`. However, if we need to enhance and encapsulate a lot of `components` then the constraints better be

$\mathcal{R}$140.

```
cannot_include(C1,_) :- wraps(C)@C1,
        not eligible_wrapper(C1,C).
```

*A wrapper must satisfy the constraints defined by the auxiliary rule* `eligible_wrapper`.

$\mathcal{R}$141.

```
eligible_wrapper(C1,C) :-
        inherits(C)@C1,
        defines(attribute(compRef),of_type(C))@C1),
        setOf(X,defines(routine(X),_)@C1,S),
        check_redefinition(S,C1).
```

*A wrapper* `c1` *for a component* `c` *must inherit from* `c`*, have an attribute* `compRef` *of class* `c` *and must (re)define the routines that are defined in* `c`*. This rule retrieves the names of all routines defined in the component class and passes them to the auxiliary rule* `check_redefinition` *(see below) below.*

$\mathcal{R}$142.

```
check_redefinition([],_).
```

$\mathcal{R}$143.

```
check_redefinition([X|Xs],C1) :-
        defines(routine(X),_)@C →
        check_redefinition(Xs,C1)|fail.
```

*The above two rules together checks whether the routines passed in as input are redefined.*

Figure 11.10: Prohibition Rules for the Wrapper

enforced rather than programmed. The property `wraps(c)` designates a wrapper that encapsulates a component of class `c`. Without loss of genericity, let us assume that the component object of a wrapper will always be attached to the `compRef` attribute of a wrapper. The Darwin-E rules that capture the regularity expressed in the above constraints is shown in Figure 11.10. Note that we cannot really enforce the constraint that the redefined routines must dispatch to the component by our rules, we can only make sure that such routines are redefined. This is one example where the desired constraint cannot be imposed as a regularity, precisely because the constraint involves the semantic specification of the routines concerned and regularity is not meant to guarantee the specification.

## 11.4 Imposing Design Principles And Its Project-Specific variations

One often adopts certain principles regarding how to design and construct a system. Such principles may reflect one's general programming style or a convention adopted by the enterprise, or a certain well-known thumb-rule. We argue that the environment that manages a software project should be able to provide a flexible support for formulating and imposing such principles. In this Section, we will substantiate our argument by showing how this can be realized for a well-known principle named, the Law of Demeter. We will begin with a brief discussion of LoD, followed by the law fragment containing the rules that formulate it. We will conclude by showing how several project-specific exceptions to LoD can be accommodated in our law. In this particular case, we will use permission based control to make the representation simpler, which means the rules presented in this section cannot be readily plugged into other law-fragments.

### 11.4.1 The Law Of Demeter

The *law of Demeter*[31] is a design principle that *rejects* certain operations from inside the methods of a class. One of the formulations of the law which lends itself to compile-time enforcement can be phrased as follows[3]:

**The Law of Demeter (Class Form):** *Inside an operation* `o` *of class* `c` *one should call operations of only the following classes (that are often called the preferred supplier classes):*

1. *the classes of the immediate subparts (computed or stored) of the current object,*
2. *the classes of the argument objects of* `o` *or the class itself,*
3. *the classes of objects created by* `o`.

In essence, if operation `o` of class `c` contains a call to another class `d`, such that `d` is not mentioned in the feature declaration part of `c`, or in the declaration of formal arguments or local entities of the operation `o`, then such a call violates the law of Demeter. As an illustration, consider the following code fragment which is a translation of the C++ code fragment in page 15 of [32] into Eiffel:

```
class BOOK
feature
     ...
     ...
end; -- class Book

class MICROFICHEFILES
feature
   search(book: BOOK):BOOLEAN is
          ...
          ...
   end;
end; -- class MicroFicheFiles
```

[3]Different formulations, with minor differences can be found in [33, 31, 32]. This particular version, obtained from Karl Lieberherr by private communication, is the most recent formulation.

```
class DOCUMENTS
feature
   search(book: BOOK):BOOLEAN is
          ...
          ...
   end;
end; -- class Documents

class ARCHIVE
feature
   m: MICROFICHEFILES;
   d: DOCUMENTS;
   search_good_style(book:BOOK): BOOLEAN is
   do
      Result:= m.search(book) or d.search(book)
   end;
end; -- class Archive

class REFERENCESEC
feature
   a: ARCHIVE;
   search_bad_style(book: BOOK): BOOLEAN is
   do
      Result:= a.m.search(book) or a.d.search(book)
   end;

   search_good_style(book: BOOK): BOOLEAN is
   do
      Result:= a.search_good_style(book);
   end;
end -- class ReferenceSec
```

The function `search_bad_style` in class `REFERENCESEC` violates the law of Demeter because it calls the `search` function on classes `MICROFICHEFILES` and `DOCUMENTS`, which do not appear anywhere in the body of the class `REFERENCESEC`. In other words, `REFERENCESEC` is not directly involved with `DOCUMENTS` or `MICROFICHEFILES`, yet the function `search_bad_style` induces a dependency on these two classes. Such dependencies are considered harmful and are not approved by LoD.

LoD controls the complexity of programming by restricting the dependency of a given class `c` to a set of classes that are "closely related" to it: the classes it refers to as *preferred suppliers*. The leverage comes from the fact that this set is a small subset of all classes in the application. To illustrate this let us consider a method `m1` of a class `c1` being renamed. Without LoD, `m1` can potentially be called from any class in the system, and therefore such a change can break the system in hard to locate places. If LoD is followed, a class `c` can call `m1` only if `c1` is "closely related" to `c`, a fact

that can be determined by examining the class `c` itself. Therefore, adapting to such changes becomes easier. Similarly, if the composition of `c1` changes (however the public interface stays invariant), LoD ensures that only `c1` and its children are affected. In [31, 32, 33] these and other benefits of LoD are discussed in great detail. LoD helps in systematizing reuse as well: when class `c` is being reused in a different context only this small subset of "closely related" classes are needed to be brought in. Then for each class in this subset one can recursively compute which other classes need to be brought in as well. However, the notion of "closely related" classes postulated as the preferred suppliers in LoD is not arbitrary; it has been shown in [32] that the calls permitted by LoD are *necessary* in the sense that if we disallow any of these calls in an attempt to reduce the dependency on other classes any further, we will not be able to write useful programs.

LoD is used by the Demeter system [22] in generating and transforming C++ code. It is not incorporated in any programming language. Conventional environments do not impose the law either. Use of this principle outside the Demeter system is therefore limited to posing it as a guideline on class design and implementation. However, as we shall see, under Darwin-E LoD and its several variations can be imposed on Eiffel software.

### 11.4.2 Formulating The Law of Demeter As Darwin-E Rules

The three clauses in the statement of LoD as presented earlier, results in four `can_call` rules, that are shown in Figure 11.11. The correspondence between the clauses in the statement of LoD and our rules is explained below.

Clause 1 of LoD deals with the *stored and computed* immediate subparts of an object. To see what it means in terms of Eiffel, consider an instance `i` of a class `c`: the attributes of `c` are the *stored* immediate subparts of `i` and the routines of `c` that return objects, are the *computed* subparts of `i`. In rule $\mathcal{R}144$, calls to classes of the features (attributes and routines) declared in `c` are permitted. This rule alone, however, may not be enough for the first clause, because of the choice whether or not the inherited features are considered to be *immediate subparts*. If we choose to consider inherited features to be immediate subparts, an interpretation which is often known as the weak version of the law of Demeter, we will need Rule $\mathcal{R}145$. This rule approves calls made to classes of features declared in ancestor classes. If we choose the other alternative, known as the strong version of LoD, which does not consider inherited features as immediate

$\mathcal{R}$144.

```
can_call(_,C1,_,C2) :-
        defines(attribute(_),type(C2))@C1 |
        defines(routine(_),type(C2))@C1.
```

*Calls made to classes of features declared in the class are allowed.*

$\mathcal{R}$145.

```
can_call(_,C1,_,C2) :-
        ( defines(attribute(_),type(C2))@C3 |
        defines(routine(_),type(C2))@C3 ),
        heirOf(C1,C3).
```

*Calls made to classes of features declared in* `C1`*'s ancestors are allowed. The goal* `herirOf(_,_)` *is defined by a built-in Darwin-E rule with obvious meaning.*

$\mathcal{R}$146.

```
can_call(R,C1,_,C2) :-
        defines(_,of(C2),as_arguments_of(R))@C1.
```

*Calls to the classes of arguments of routine* `R` *are allowed from* `R`.

$\mathcal{R}$147.

```
can_call(R,C1,_,C2) :-
        defines(_,of(C2),local_to(R))@C1.
```

*Calls to classes of local entities are allowed.*

Figure 11.11: Rules That Impose LoD

subparts of an objects, we will not need Rule $\mathcal{R}145$.

Clause 2 of LoD allows calls from a routine `r` of class `c` to `c` itself and the classes of the arguments of `r`. Rule $\mathcal{R}146$ allows calls from a routine to the classes of its arguments. In Darwin-E calls to a class itself are always legal, therefore we do not need to worry about such calls in our rules.

Clause 3 of LoD allows calls to the classes of objects created by a routine. In Eiffel remote creation is not allowed, a routine `r` of class `c` can create only local entities or attributes of class `c`. Classes of attributes are already dealt with in rule $\mathcal{R}144$, and rule $\mathcal{R}147$ permits calls to the classes of the local entities of `r`.

### 11.4.3 Project-Specific Exceptions To The Law of Demeter

Unfortunately, uniform imposition of LoD is not always desirable. Like other broad principles, it is not always useful, and is sometimes detrimental. In this section we describe several such situations, and show how LoD can be relaxed to accommodate them by introducing exceptions in the law of the system. We conclude this section with a brief note on how the process of admitting exceptions is controlled under Darwin-E.

#### The Case of Transient Classes

If a class is used only for a small transitory period of time, its dependencies on others are not important, because we will neither reuse it nor maintain it as the other classes evolve. Classes built for testing a certain feature or an idea are examples. One might want to relax LoD so that it does not apply to such classes. This can be done as follows.

Given a class `c`, which we consider transient, one define the following additional rule:

$\mathcal{R}148$.

```
can_call(_,c,_,_) :- true.
```

This rule simply permits an individual class `c` to make any call.

One can also systematize this concept of *transient* classes by characterizing all classes considered transient by a property `status(transient)`. In this case one can relax LoD for all these classes by the following single rule:

$\mathcal{R}149$.

```
can_call(_,C,_,_) :- status(transient)@C.
```

### The Case of Stable Classes

A certain set of classes in a project may be very stable and unlikely to change. Examples of such classes are the classes in the installation library or classes that contain legacy code. Constraining calls to these classes by LoD does not serve any purpose since the callee classes are not likely to change and dependency on such classes is therefore not harmful as far as LoD is concerned. One can characterize all such classes by means of the property `status(stable)`, and add the following rule to the law of the project, in order to admit all calls made to a *stable* class:

$\mathcal{R}$150.

```
can_call(_,_,_,C) :- status(stable)@C.
```

Classes in the installation library are a special case of these *stable* classes. In Darwin-E, any class from the Eiffel installation library that is used in the system, is already characterized by the built-in property `library`. This property can be utilized in admitting all kinds of calls to library classes from any class in the system in a manner similar to rule $\mathcal{R}$150:

$\mathcal{R}$151.

```
can_call(_,_,_,C2) :- library@C2.
```

### The Case of a Close-Knit Cluster

Some clusters of classes may have a very cohesive structure with strong interdependencies. In this case preventing classes in the same cluster to call each other just because they violate the law of Demeter may not make sense, since any change in one class is likely to change all the classes in the cluster anyway. Moreover, the prohibition of LoD on inter-cluster calls can be particularly annoying in this case, because they are likely to be used very often and by the programmers (the ones who designed and maintains this cluster) who know what they are doing. In this situation, one might include the following rule in the law:

$\mathcal{R}$152.

```
can_call(_,C1,_,C2) :-
       cluster(X)@C2,
       cluster(X)@C1.
```

This rule exempts the calls made by a class to another class in the same cluster from LoD.

### The Case of Acquaintance Classes

The notion of *acquaintance classes* of a method was proposed in [32]. A class `c1` may have an unusually close relationship with a class `c2`, although in a routine `r` of `c1`, calls to `c2` do not satisfy LoD. In this case it makes sense to allow such calls. To incorporate this we may use the following rule:

$\mathcal{R}$153.

```
can_call(r,c1,_,c2) :- true.
```

This notion can be extended to make `c2` an acquaintance of all routines of `c1` as follows:

$\mathcal{R}$154.

```
can_call(_,c1,_,c2) :- true.
```

or to have multiple classes `c3, c4` and `c5` as acquaintance of `r`:

$\mathcal{R}$155.

```
can_call(r,c1,_,C) :- C= c3| C= c4| C= c5.
```

### A Concluding Remark: Controlled Relaxation

So far we have seen that under Darwin-E, LoD can be relaxed by adding `can_call` rules to the law of the project. Such rules provide a precise definition of the exceptions to LoD that are being admitted. This scheme of relaxing a principle such as LoD can be meaningful only if there is a strict control over creation and destruction of the individual rules. Otherwise, anybody can bend the law at will, simply by adding new rules or destroying existing ones. Manipulation of user-defined properties should also be strictly controlled, because one can evade the law by adding or removing the property used in existing `can_call` rules. As we have explained earlier, these aspects can also be controlled by law under Darwin-E.

# Chapter 12
# Regularities That Prescribe A Desired Behavior

So far our regularities have been mostly concerned with deciding whether *something* (such as an inheritance relationship between instances of two classes, a call taking place between two objects) *can* or *cannot* happen. However, there are certain kinds of regularities that *prescribe* something to happen. Such regularities can be incorporated in our scheme by expressing the intention in our rules and performing the required code transformation during law enforcement. Many such regularities, demanding different kinds of code transformations, can be envisioned, although we have considered only a few, just to demonstrate the versatility of our regularities. Note, however, that this code transformation approach to prescriptive regularity applies only to cases where source code is available, and in all the examples in this chapter we will assume that it is.

## 12.1 Forcing Inheritance

Suppose we want a set $S$ of objects to possess a certain characteristics. This can be done by encapsulating the desired characteristics into a class `c` and forcing the classes that generate objects in $S$ to inherit from `c`. A case in point is the situation we encountered earlier in Chapter 9, where we needed a special object `spy` to be available to certain objects at run time, which we solved by requiring certain classes to inherit from a designated class.

Contrary to the case of a single class `c1` inheriting from another class `c`, a constraint like the one above cannot easily be built-in to the system, specially so, when the classes may change. The only way to have a reasonable guarantee that all such classes inherit from `c`, is to perform manual checking, which is inherently error-prone, unreliable and impractical when classes change very often. Towards this end Darwin-E provides a means for inflicting forced inheritance on classes in the form of `must_inherit` rules. The effect of this rule can be described as follows:

**Definition 15 (must_inherit)** *If there is a* `must_inherit` *rule that makes the* `must_inherit(c1,c2)` *predicate succeed, then the code attached to* `c1` *is transformed to make* `c1` *inherit from* `c2`*. If* `c1` *already inherits from* `c2`*, no action is taken.*

This rule is considered when code is being attached. The transformation, if necessary, is performed at the same time, and upon the very code that is being attached.

Let us now recall the example project in Chapter 10. Let us assume that we now require that the special object spy should be available to instances of all classes in the base division. The following rule can be used for this purpose:

$\mathcal{R}$156.

```
must_inherit(C1, C2) :- division(base)@C1
       className(spylocator)@C2.
```

As another example, consider the well-known design of having an abstract class for each type of product objects to be used in the system. For instance, we will have a WINDOW class to define a generic window, and there will be many specific products such as a motif window (an instance of `MOTIF_WINDOW`), a open-look window (an instance of `OPEN-LOOK_WINDOW`) and so on. It is self-evident that each specific product class ( such as `MOTIF_WINDOW, OPEN-LOOK_WINDOW` ) must inherit from the generic product class (which in this case is `WINDOW`), but it is a formidable task to make sure in a large project that all the specific product classes that will ever be written will inherit from their corresponding generic product class.

A rule like the following, can help in this situation:

$\mathcal{R}$157.

```
must_inherit(C1, C2) :- specificProduct@C1,
       genericProduct@C2,
       productType(X)@C1,
       productType(X)@C2.
```

The above rule uses properties that were already introduced and explained in Section 11.3.2.

## 12.2 Killable Objects

In many applications, it may be needed to *kill* an object, where the notion of being *killed* can be described as follows. An object behaves normally unless and until it is *killed*; a *killed* object is essentially thrown out of the active computation by force. We call the objects that can be used this way the *killable objects*. Killable objects are essential in data-oriented applications where objects may become *corrupt*, *invalid* or in modeling and simulation applications where we need to model *failure*, *fault*, *downtime*, etc.

Eiffel does not provide for object destructors for reasons such as safety against dangling pointers etc. In the absence of destructors, an expanded entity `e` can never be physically removed from a system. The object attached to a referenced entity `r`, on the other hand, may still be around before being garbage collected (and have other live references) even if the reference `r` is voided ( by an assignment `r:= void`). This means that killable objects must be supported at the application level.

The killable objects reflect a pattern, whose essentials are as follows.

- All classes that give rise to killable objects:
  - should maintain the state of `killed` or `alive`.
  - should be equipped with the methods `kill` and `revive` to activate the transition between `killed` and `alive` states.
- All the other methods in such classes:
  - should look at the state before execution; if `alive` it would do what it is supposed to; if `killed` it would produce an error message.

The universal quantifiers in the above principles suggest that this pattern should be imposed as a regularity.

The two clauses of the first item can be easily imposed on a set of classes by forcing them to inherit from a class `KILLABLE` whose definition is as follows:

```
class KILLABLE
  feature
   kill is
   do
     killed:= true
   end;

   revive is
   do
    killed:= false
   end;
 feature {NONE}
   alive:BOOLEAN is
   do
     Result:= not killed
   end;
   error is
   do
    io.output.putstring(``invoked a killed object\n'')
```

```
    end;
    killed: BOOLEAN
  end
```

Note that, if class KILLABLE is simply inherited in a class c, instances of c

- are created in the alive state, because the default value of the BOOLEAN variable killed is false, and
- can be freely transformed from killed state to alive and vice versa by invoking the procedures revive and kill respectively.

In the previous Section we have seen how classes can be forced to inherit from a specific class by creating a must_inherit rule. If the property killableClass is chosen to denote the classes that give rise to killable objects, the following rule will suffice in this case:

$\mathcal{R}$158.

```
must_inherit(C1, C2) :- killableClass@C1,
       className(killable)@C2.
```

The desired behavior of killable objects demands that each routine r of the form:

```
r( .. formal_parameters_of_r...):.... return_type_of_r is
   do
         [[ actual_body_of_r]]
   end
```

defined in a killable class c, must be transformed into:

```
r( .. formal_parameters_of_r...):.... return_type_of_r is
   do
      if alive then
         [[ actual_body_of_r]]
      else
         error
      end;
    end
```

Darwin-E provides a mechanism, which transforms the code of a class c in cfg's directory accordingly. This mechanism is encapsulated by the built-in make_killable rule, which can be used as follows:

$\mathcal{R}$159.

```
can_include(C1,_) :- killableClass@C1,
       make_killable(C1).
```

The above rule ensures that the classes designated by the killableClass property will

be transformed appropriately, in order to be used in any configuration.

Note however that a class `d` does not become killable class simply by inheriting from another killable class `c`.

Such inheritance equips class `d` with the methods and attributes defined in class `KILLABLE`, but routines defined in `d` (which includes redefinition of routines inherited from `c`) may not be of the desired form. This is because a rule like $\mathcal{R}159$ above does not consider and transform the children of a killable class. As a result, instances of `d` that can be polymorphically assigned to instances of `c`, can bring about undesirable effects on the clients that expect killable objects. This, however can be easily fixed by reformulating the `can_include` rule as follows:

$\mathcal{R}160$.

```
can_include(C1,_) :- (killableClass@C1|
       (heirOf(C1,C2),killableClass@C2))
       make_killable(C1).
```

The above rule makes sure that the classes designated by the `killableClass` as well as any descendant of such classes are considered by the `make_killable` rule.

# Chapter 13

# Related Work

The idea of imposing constraints on software systems is not new. It appeared in several incarnations in different systems and different contexts, but so far, it has never been used the way we used in the context of Darwin-E to formulate and impose *regularities*. We will describe a few such systems, where similar attempts has been made, but as we shall see, very few systems are capable to capture the essence of our regularities. There are a bunch of (process-centered) environments such as Marvel [3], Adele/Tempo[4], Merlin [23] etc. in which *enforcing some kinds of constraints, either prescription or proscription, on the software process* is an important objective. The modality and scope of the constraints vary in different environments. Most of these existing process-centered software development environments deal with C or C++ and stress more on process modeling, process enactment, intelligent assistance and guidance. We will also consider two representative samples to contrast with Darwin-E.

## 13.1 ADA9X

Attempts to support software engineering constraints can be observed in ADA9X [69] language, by means of various pragmas that are interpreted and enforced by the compiler and in *Ana*, a language for annotating ADA programs [36]. As an example, by utilizing the built-in *pure pragma*, ADA functions can be rendered side effect free, as we do with our SEF routines. However not all the regularities can be enforced this way, because some of them may involve aspects of a project that are simply beyond the scope of the programming languages. For instance, the constraint that *modules owned by trainee programmers cannot access hardware directly*, is hard to formulate by means of a pragma or by using ANA annotations.

## 13.2 CENTAUR

In an earlier system named CENTAUR [12], the syntactic and semantic specification of a programming language was used to derive a language specific environment. Some of the constraints we impose by our regularities can be expressed as part of the static semantics of their target environment; however, the semantics of their specification language TYPOL does not seem to interpret global properties such as the notion of clusters, nor express constraints based on these properties.

## 13.3 CCEL

An independent effort to constrain the structure of object-oriented programs can be found in the *C++ Constraint Expression Language* (CCEL) [1, 38, 13] and its host system. In essence, the CCEL system is a tool that can formulate a certain class of regularities and check for violations of these regularities in C++ [68] programs. The range of regularities that can be handled by CCEL is limited in two major ways: 1) CCEL does not yet handle interactions that originate from the bodies of C++ functions and procedures such as *call*, *assign* or *creation* and 2) CCEL does not recognize any properties of classes that are not part of the C++ language. Perhaps the most important difference between CCEL and Darwin-E is that CCEL is a tool whereas Darwin-E is an environment that provides an uniform control over other aspects of the project including its evolution.

## 13.4 Reflexion Model

There are other systems with a different orientation but somewhat similar approach as our regularities. In [54], analyzing the source code to construct a *reflexion model* is proposed, in order to check whether a system conforms to a high-level model. A reflexion model and associated tools can be used in understanding and reverse engineering existing systems. If desired, Darwin-E can similarly be used as a reverse engineering tool to check whether a given system does possess a given set of regularity.

## 13.5 Pattern-Lint

In pattern-lint [65], static analysis and dynamic visualization is used to monitor whether a software system complies with its high-level design model: a database of implementation-level relations are created from the given program entities, which is then matched against the rules of a given design model to produces relations that demonstrate consistency with or divergence from the design model. This is very similar to our approach, but we do not consider dynamic visualization, and pattern-lint relies on a library of design models. Under Darwin-E, various design models can be constructed by identifying the underlying principles and architectural patterns and formulating them as regularities.

## 13.6 Module Interconnection

Since the introduction of the concept of module interconnection [11], there has been several successful attempt at specifying constraints over the interconnection between the various modules of a system. Three prominent ones are the PIC formalism of Wolf et al. [72], the Inscape system of Perry [58] and the work on connectors by Garlan and Shaw [18, 66]. Although they represent very powerful techniques for specifying interfaces between modules, and in many ways they can do what we cannot do in Darwin-E, they usually are not concerned with regularity, i.e., statements involving universal quantifications. For instance, none of these techniques can specify and guarantee that a given system is layered, the way we showed in Chapter 11. Incidentally, in [55, 56], a mechanism for imposing a layered module interconnection structure on a given system was described. Although this bears certain similarity to our approach to impose layered design, it is not general enough to impose structures other than layers.

## 13.7 Demeter and Adaptive Programming

The basic idea behind Demeter methodology [31, 32](also called *adaptive programming*) is as follows. Small and loosely coupled methods (in classes) are considered good software engineering practice because they are simple, they enhance reusability and are easier to maintain. It has been observed that most of the small routines in an object oriented program act as dispatchers; the actual work that gets performed is fairly localized. Therefore an object-oriented program can be viewed as a traversal of a complex

object structure. The Demeter approach has identified a number of such *propagation patterns* and developed a language to specify them: given a class hierarchy and a propagation pattern for a task, the corresponding Object-Oriented code is automatically generated by the Demeter tools. This has several advantages all of which are primarily due to the separation between the actual work and the traversal pattern, which in turn means that the class structure can evolve independently. Of course the usefulness of this approach depends on 1) how much traversal really happens in practical o-o systems and 2) can all possible traversal patterns be specified by this approach.

In some sense, Demeter approach imposes a certain kind of regularity, namely the propagation patterns, upon object-oriented software. The nature and mechanism of imposition of such regularity differ significantly from ours, much of the propagation pattern has to be coded in the Demeter language which will be interpreted by Demeter tools to produce C++ code that conforms to the specified pattern. The Demeter methodology depends crucially on several design principles such as the Demeter's Laws. We have already shown that such laws can be formulated as our regularity.

## 13.8 Marvel

Marvel [2, 3, 5, 24] is a multi-user process-centered environment. Its main objective is to define a process (software project) by means of its rule-based process modeling language (PML) and support process enactment and process evolution (changes made to itself during its execution). The rules used in Marvel to define a process are of two types: *consistency* rules that ensure the consistency of the object-base ( collection of attributed instances of Marvel classes) of the project and *automation* rules that perform activities in the object-base. Each activity corresponds to a user command and is encapsulated by a condition that must be satisfied before initiating the activity and an effect that asserts the result of completing the activity on the state of the process. Marvel allows the user to turn off or on backward and/or forward chaining of automation rules, but consistency chaining is mandatory: an activity can be carried out only if all consistency predicates in the precondition of the rule fired by it are true and all consistency predicates in the effect of the rule are asserted.

In Darwin-E, activities correspond to commands, but unlike Marvel, the semantics of Darwin-E commands are built-in; users are not responsible for writing the shell script envelope that encapsulates the activity in a rule. Darwin-E's object base consists of attributed instances of classes, but unlike Marvel multiple inheritance is not supported.

Darwin-E does not incorporate rule chaining at all and Darwin-E rules are consistency rules in Marvel terms. Darwin-E governs user commands; in this sense it provides *guidance* and *assistance* to some extent, but a large amount of software process activity relating to design and analysis is assumed to be carried outside of Darwin-E. Therefore, Darwin-E does not even come close to the level of process management offered by Marvel. This is so because Darwin-E is not meant to be a process centered environment and process modeling and management is not our goal. Imposing regularity on the product is an important objective of Darwin-E, which is not so in Marvel.

## 13.9 Adele/Tempo

The Adele/Tempo [4] environment uses a kernel based on an entity-relationship database, complemented by object-oriented concepts and an activity manager based on triggers. In this environment, objects play different roles in different work environments, a connection with the work environment establishes the dynamic and context sensitive behavior of the objects. A process model is specified as a combination of software process types. A software process type is a standard object type in the underlying database, therefore process types can be redefined or specialized, it could be instantiated and an instance could be connected with other objects. A process type identifies and describes a set of activities. A software process is a set of these activities executing concurrently and asynchronously. All manipulation of the objects generate events, and the event-condition-action (ECA) triggers defined in the process types take these events into account in their respective work environments.

Our rules in Darwin-E, in some abstract sense, can be thought of as ECA rules: issuing a command is the *event*; if the law approves it, *actions* associated with the command are carried out. Like Darwin-E, Adele/Tempo imposes some constraints on the product too. These constraints are integrity constraints on the product model, expressed again in terms of ECA rules. While these rules may capture the global nature of our regularities on the product, Adele does not try to impose them on inter-component interactions like inheritance or being a client as done in Darwin-E. The constraints on the product model in Adele/Tempo results in a sophisticated configuration management. Darwin-E also has its own consensus based configuration management system which is discussed in Chapter 8.

# Chapter 14
# Shortcomings and Future Work

The work presented here is not without its share of limitations and openings for further research. We will discuss the shortcomings first and then talk about future reserach possibilities.

## 14.1 Shortcomings

The expressive power of regularities as described in this dissertation is limited by how we store the results of our static analysis. Although we construct an abstract syntax tree, we retain only some parts of it that are optimized to generate the interactions we handle. This means that Darwin-E cannot handle regularities for which we do not keep enough information. For example, the definition of our call interaction does not include the actual parameters of the call, therefore if we want to impose a constraint involving the parameters, such as *the first parameter of all calls to the kernel cluster must be the special variable* `this`, cannot be enforced. However, by keeping a representation of the full abstract syntax tree of the Eiffel system, we can practically avoid all such problems.

In general, the notion of regularity can be used to express constraints upon the semantics or behavior of the system. In our treatment of regularity we cannot capture semantics very well. For instance, recall that one of the constraints inherent in the wrapper design pattern is that *the wrapper class must redefine inherited methods to dispatch to the component object it encloses.* This is a constraint involving the semantics of the redefined features, and as we mentioned earlier cannot be expressed by our rules. Furthermore, constraining the behavior of a system may involve runtime actions which are prescribed in the ruling and our approach (discussed in Chapter 12) to prescriptive regularities is not general.

We chose to impose the regularities at compile time. This, in the presence of the dynamic binding and polymorphism provided by Eiffel, means that the call and assignment interactions may occasionally be mis-treated by our regularities. We have

pointed out this problem in Chapter 6. More generally, we do not have proper means to infer run-time state of the entities, and our control is limited to what we can decide at compile time. However, this is inherent in any kind of static analysis.

In LGA, and therefore, in Darwin-E users are allowed to formulate various rules to express the desired regularity. It is possible that users may create inconsistent rules – rules that contradict each other, or they may forget to create some desired rules – making the law incomplete. The completeness issue is handled by carefully designing a set of built-in rules that are guaranteed to be complete (i.e. Prolog will find an answer), and then leaving some hooks as the only rules that can be defined by the users. The consistency issue in its full generality is a hard problem and we attempted to address it by controlling rule creation activity. We assume that a small set of responsible people will design the law, and define what other rules can be created in the project and by whom.

In our current implementation, an already enforced Eiffel system is re-analyzed and re-enforced in its entirety, even if only a small part is changed. A carefully crafted incremental analysis and enforcement mechanism is needed, because the analysis and enforcement is expensive as we discussed in Chapter 6.

The Darwin-E prototype is somewhat slow at times. The reason is that the entire environment, including the X interface and the TCP/IP clients and servers are implemented in Prolog. We used Prolog because of its ease in prototyping and because it leads to easy representation and management of the law. But Prolog code is slow, and there are some situations where this shows up glaringly. The Eiffel front-end that performs the static analysis of a configuration to identify Eiffel interactions and the static enforcer that imposes the law on these interactions are cases in point. The latter is a major efficiency bottleneck of Darwin-E.

The Darwin-E environment is not very scalable. We have implemented the object-base and various algorithms for managing the object-base (such as call backs, locks) in Prolog. This loads the Prolog runtime database, which has a physical limit (The maximum size of Prolog's data area is 1 GB, which must come from the low 1 GB of virtual memory.). This is not sufficient for a large project.

Finally, the Darwin-E prototype is not very user-friendly. It could definitely use some more on-line help and tools to assist the users.

## 14.2 Further Research

We have used regularities in a rather ad-hoc manner to prescribe certain interactions by law and instrument the code accordingly before compilation. One could find many other examples of such situations. While we are able to demonstrate that this can be done, a lot of questions remain un-addressed. For example, is there any commonality among these situations that can be generalized to a simple framework? Can this technique be used to impose meta-object protocols in a manner similar to other compiler-based approaches such as [25]?

The desired regularities in a project are currently formulated in the project and the rules that express them are created in the project. But this view is not compatible with the recent trend of *reuse*, and *component based* software construction technologies. Components are not necessarily self-complete systems and most certainly are not individual classes, which are expected to be reused across projects, and to complicate things for us, the components are often binaries– compiled object code. Components may have regularities associated with them, regularities that impose certain internal structure within the component as well as control how the component should be used in a global context along with other components. So when a component is brought in for reuse in a project, such regularities must also be brought in. Our methodology should be adapted to this emerging technology. We hypothesize the following component oriented view of software. A software component is a bunch of classes with certain rules and guidelines. Rules embody the regularities that we can enforce and guidelines are the tasks the users are supposed to perform. We believe this is one way to tackle the problem of *architectural mismatch* as documented by Garlan et al. in [17], in the context of reusable component oriented software construction.

Use of law-governed obligation has been demonstrated elsewhere [39]. But we have not investigated obligation in terms of Darwin-E and regularities in object-oriented software at all. This area remains open for further research. In particular, the process aspect of Darwin-E can benefit significantly from obligations.

Regarding the Darwin-E environment itself, one immediate objective would be to improve the efficiency and scalability of the environment so that it could be commercially viable. We have studied several public domain Eiffel front-ends (such as ep[1]), and found out that such parsers (usually written in C) could lead up to several degrees

[1] ep is a front-end for Eiffel, part of the CockTail toolbox, developed at Karlshrue

of speed-up for the static analysis of Eiffel code. This alone would not mean much because the major bottleneck is the static enforcer. An alternative efficient formulation of law is yet to be researched.

Our immediate goal is to try to realize a our system in terms of the Unix tool `make`. Like a `Makefile`, this proposed tool will take a file containing the system description as an input. The description will contain classes, the properties associated with them and the rules about the system. The tool will first perform the static analysis (and create the abstract syntax tree) and then check the rules (by running a program on the tree). If there is no violation then, it will start compilation.

Our regularities originate at the design and architecture level, and we impose them at the code level. While this is clearly important in establishing regularities as invariants of evolution, possibilities exist that the imposition can be done at the design-level. We would like to explore both domain-specific and domain-independent design languages, if it is possible to impose our regularities upon designs expressed in such languages. The possibility of tools that compile such designs into executable code makes such such an approach look very promising.

# Chapter 15
# Conclusion

We showed that regularities, which are argued to be essential ingredients of large software systems, can be formulated and enforced on object-oriented software. The Darwin-E prototype, which allows us to impose regularities upon Eiffel system, was developed for this purpose. Various useful regularities are demonstrated using this environment. However, we do not claim that regularity is the *silver bullet*; in fact, use of regularities in the engineering of large systems can only *reduce* (as opposed to *eradicate*) certain inherent problems like complexity, malleability and invisibility.

The Darwin-E prototype is a *proof of concept*, rather than a commercial product. However, we understand what is needed to make it commercially viable and have envisioned a future product. We have designed Darwin-E in such a way that one needs to change only the language interface to handle, for example C instead of Eiffel.

In addition, Darwin-E is a product centered environment. It does not aim to model or enact software process, its main objective being to impose regularities. It provides only a minimal control over software process, along with certain other facilities that can benefit from the underlying LGA. We feel that it is important to focus on the product and we demonstrated achievable benefits by means of Darwin-E. Recent surge in product-centric research also corroborates our point of view.

Finally, this work also validates the usefulness of LGA, which provides a framework for formulating and enforcing artificial laws about software systems and its development processes.

# Appendix A
# Summary of Implementation Details

The Darwin-E environment is based on a client-server architecture: each project under Darwin-E is given a dedicated server that manages the project object-base. Human users involved in the project interact through shells that are clients to the project-server. The client-server communication is implemented using TCP/IP. Each client shell also has an X based message window that displays various diagnostic messages from the server.

The server is responsible for managing the object-base. Each active client shell is associated with one builder object in this object-base. Therefore whatever an user sitting at the client shell does is reflected as an interaction initiated by the corresponding builder object in the object-base.

From the top level, the server an be viewed as a composition of the following four major subsystems:

- **message handler and dynamic enforcer:** This subsystem is responsible for intercepting user commands issued as messages, computing the ruling for it with respect to the law, and carrying out the ruling with the help of the other subsystems.

- **eiffel manager:** This subsystem is responsible for carrying out operations that involve the underlying Eiffel environment. For example to invoke eiffel tools like *ebench*, or *flatshort*. Often, services of this subsystem are used by the configuration manager, as explained below.

- **configuration manager:** This subsystem is responsible for managing configuration related activities in the server. When a user creates a configuration, this subsystem is responsible for creating the configuration object, its associated directory and Ace files. Often this subsystem uses services of other subsystems. For example, when a configuration is being assembled it uses the eiffel manager to invoke the eiffel compiler. When regularity is imposed upon a configuration, it

uses the services of the language front-end and static enforcer as follows.

- **language front-end and static enforcer:** This subsystem is responsible for the static analysis of the classes in a configuration to identify the various interactions and enforce the law upon these interactions.

Besides the above major subsystems, there are a few other subsystems such as the `lock manager` that takes care of Darwin-E's locking mechanism. These subsystems are in a lower level than the principal subsystems discussed above. At the core of the server, a basic kernel implements primitive operations such as creating and destroying builders, modules or rules. The core is used by all the subsystems.

# Appendix B
# Optimizations For Static Analysis

As we have explained in Chapter 6, the way we impose regularity is a time consuming process. The major bottle neck is enforcing the law on each and every interaction. To reduce the load on the static enforcer, we adopted the following guideline: *do not consider an interaction unless absolutely necessary.* This lead to several simple optimizations policies that we have incorporated into our static analyzer. They are described as follows:

- Do not consider any interaction if the Law is empty.
- Consider an interaction $\mathcal{I}$, only if the Law intends to control such interactions. For example, if the law does not contain any `can_inherit` or `cannot_inherit` rules, there is no need to consider inherit interactions. However, there are certain Eiffel constructs as discussed in Chapter 6, that can be considered as multiple interactions. For example, object generation is also considered as an assignment interaction. For such cases, we must consider the interaction if the Law contains *any* rule that is relevant for it. For example, in case of a `!!x` construct the static analyzer must check if there is any prohibition or permission for *assign* as well as *generation* exists.
- Consider an interaction $\mathcal{I}$ only if the interaction from the current class is relevant with respect to the Law. Even if it contains a `cannot_inherit` or `can_inherit` rule, the law may be *indifferent* to the *inherit* interaction of a particular class `c`. It is possible to find out while analyzing a class `c`, whether a certain interaction $\mathcal{I}$ from `c` is relevant with respect to the law or not. We consider the interactions only if it is so.

Our experiments show that these optimizations cut about 10-15 percent interactions from our considerations in a typical application involving 10 user defined classes and a moderately complex law.

# Appendix C
# Glossary of Commands

## C.1 Connection Commands

1. **`wcli:`** The command **`wcli`** at the unix prompt starts a shell that will act as a client to the project-server. The user and his current directory will then be registered with the Darwin-E environment. In response, he will be prompted for further actions and a message window in which various diagnostic messages from the server will be displayed, will appear in his screen.

2. **`activate(p) :`** In order to connect to a project **`p`**, a user needs to type in **`activate(p)`** at the prompt in the shell started by **`wcli`**. If **`p`** is a new project, unknown to the environment or an old project, saved but not currently active, Darwin-E will set up a project-server for **`p`** on a machine across the network - defined in the environment *a-priori*. (The builders usually would operate from machines different from this machine on which the project-server runs.) If **`p`** is a brand new project, that is, the environment has just created a new server for it, the user is logged in as the built-in builder **`#creator`** that has the power to boot the project-server. Booting the server is done by executing the Darwin-E script **`init3`** (see below ) at the **`#creator`** prompt. After booting up, the server puts the project in its initial state with only the built-in objects and rules. On the other hand, if the **`activate`** command has re-started the project-server for a saved project, the user will be logged in as the built-in builder **`#newcomer`** and the saved state is restored as the current state of the project. Similarly, if **`p`** was the name of an active project, that is a project-server for **`p`** was already running, then also the user would be logged in as **`#newcomer`**.

3. **`^enter(password)`** $\Rightarrow$ **`b:`** The user has to associate himself with his role of a builder **`b`** in a project, by using this **`^enter`** command if he is logged in as **`#newcomer`**. A prompt identifies the builder which the user is associated with.

4. **`exec(s):`** at the builder's prompt executes a script **`s`** (which is stored as a prolog file **`s.pl`**).

5. **`save:`** saves the current state of the project on disk.

6. **`ctrl-D:`** disconnects the client shell from the project-server.

## C.2 creation Commands

1. `^new(id,c)` ⇒ `a:` where `a` is module-object. It creates a module-object named `id` with attribute `className(c)` set in it. This object will be used to represent the eiffel class `c`. The sender becomes the 'creator' of `id`.

2. `^new(id)` ⇒ `b:` where `b` is an existing builder-object. It creates a builder-object named `id`. By default, the initial password of a builder named `id` is the name of the builder.

3. `^new(id,d)` ⇒ `aConfiguration:` creates a configuration named `id` with directory `d` as its site. A directory (if does not exist already) `d` is created, an `Ace` file is customized in this directory so that it acts as the root-cluster of the eiffel universe for the system that the object `id` represents. An attribute `directory(d)` is set in the new-born configuration object `id`.

4. `^new(id,c)` ⇒ `aGroup:` creates a group-object named `id` meant for the different incarnations of the class group-object named `id` meant for the different incarnations of the class named `c`.

## C.3 Observer Commands

1. `^more` ⇒ `o`: invokes unix more on the .e file of `o`, if `o` is a module-object. This is a way to see the body of a class object.

2. `^print` ⇒ `o` : prints all the attributes of the the object `o` on screen.

## C.4 Modifier Commands

1. `^emacs` ⇒ `o`: where `o` can be either a module-object or a configuration-object. If `o` is a module-object, Darwin-E opens an emacs window with the .e file associated with `o`.This is used to edit the body of a class. It opens an emacs window with the .eiffel file in the directory associated with `o` if `o` is a configuration-object, which is how one should edit the Ace file associated with an eiffel system.

2. `^attach(filepath)` ⇒ `o:` sets the file pointed to by `filepath` as the body of `o`. The file must define the class whose name is stored in `o` as the `className(_)` attribute, otherwise attachment fails. The attachment process also sets up the system defined interface related attributes (described earlier).

3. `^destroy` ⇒ `o :` destroys the object `o`, `o` could either be a module-object, a builder-object or a configuration-object. If `o` is a module-object and not currently included in any configuration, this object is removed from the object-base, the body of `o`, ie the .e file is however not removed from the file system. If `o` is a configuration-object, this object is removed from Darwin-E's object-base, none of the component class

objects are removed though. The directory associated with the configuration is removed. In case of a builder-object, it is simply destroyed.

4. `^set(attribute)` ⇒ `o :` sets an attribute 'attribute' in `o`.

5. `^recant(attribute)` ⇒ `o :` retracts an attribute 'attribute' in `o`.

6. `^enroll(c)` ⇒ `g :` enrolls the module-object `c` in the group represented by group-object `g`. If `c` represents a class different from the class for which `g` is meant for the message fails. Upon enrollment, `c` is noted to become the current member of `g` and the time `c` is enrolled, is also stored in `g` as the `engrouped(c,time)` attribute.

7. `^withdraw(c)` ⇒ `g :` withdraws the module-object `c` unconditionally from the group `g`. No effect if not presently enrolled in the group.

8. `^persist` ⇒ `o:` makes the object `o` persistent.

9. `^unpersist` ⇒ `o :` makes the object `o` unpersistent.

10. `^chpasswd(new_passwd)` ⇒ `o :` sets `new_passwd` as the new password of the builder-object `o`.

11. `^trace` ⇒ `o :` starts tracing out bound messages from `o` if the object `o` is a builder and starts tracing in bound messages if `o` is any other kind of object.

12. `^untrace` ⇒ `o :` turns off any active tracing on `o`.

13. `^lock` ⇒ `o :` locks the object `o` if it is not currently locked by anybody else. A composite object is locked only if all its sub-objects (recursively) can be locked.

14. `^unlock` ⇒ `o :` releases the existing lock on `o`.

## C.5 Eiffel Commands

1. `^flatshort(o)` ⇒ `cf:` invokes the tool *flatshort* on class represented by `o` with respect to configuration `cf`'s universe.

2. `^ancestors(o)` ⇒ `cf:` invokes the tool *ancestors* on class represented by `o` with respect to configuration `cf`'s universe.

3. `^descendants(o)` ⇒ `cf:` invokes the tool *descendants* on class represented by `o` with respect to configuration `cf`'s universe.

4. `^senders(o,f)` ⇒ `cf:` invokes the tool *senders* on class represented by `o` and its feature `f` with respect to configuration `cf`'s universe.

5. `^eiffelclean` ⇒ `cf:` invokes the tool *eiffelclean* on configuration `cf`.

## C.6 Configuration Commands

1. `^put(L)` ⇒ `cf`: the parameter L of this message is heavily overloaded in Darwin-E. L could be 1) a module-object or 2) a list of module-objects or 3) a (group-object, _) pair or 4) a list of such (group-object,_) pairs or 5) a list containing module-objects or pairs of the form (group-object,_). The _ denotes an unspecified variable name and `cf` is an existing configuration. It puts the module-objects (if any) in `L` into `cf` and copies the bodies of these module-objects into the configuration-directory. It sets up the baseline and system-model of the configuration appropriately (see the paper on configuration management for details). In this paper we used `put` in conjunction with a list of module-objects only.

2. `^delete(L)` ⇒ `cf`: is the functional inverse of put.

3. `^bind` ⇒ `cf`: starts up the mechanism of consensus-based configuration binding which we will discuss later. In short, as a result of this binding, a module-object for each group that was put in the configuration is chosen and its necessary files are copied to the configuration directory or if the binding fails the failure is reported.

4. `^enforce` ⇒ `cf`: enforces the law of the project on the configuration `cf`. If the enforcement is successful, i.e., there is violation of the law, then Darwin-E permits assembly of the configurations.

5. `^freeze` ⇒ `cf`: *freezes* the system represented by the configuration `cf`.

6. `^melt` ⇒ `cf`: *melts* the system represented by the configuration `cf`.

7. `^finalize` ⇒ `cf`: *finalizes* the system represented by the configuration `cf`.

8. `^run` ⇒ `cf` : runs the eiffel system represented by the configuration `cf`, provided the configuration was assembled using one of *freeze, melt* or *finalize*.

9. `^ebench` ⇒ `cf`: launches the `x` based tool named ebench on the universe of the configuration `cf`.

## C.7 Legislative Commands

1. `^createRule(r,(h:-b))` ⇒ `mR:` creates rule named `r` from the metarule `mR`.

2. `^createMetarule(m,(h:-b),L)` ⇒ `mR:` creates the new metarule `m` from the metarule `mR`.

3. `^removeRule ⇒ r:` where `r` is a rule or a metarule, removes the object `r`. Darwin-E has some built-in rules and metarules, no builder can remove them. The scope of `^removeRule` excludes such 'non-destructible' rules.

## C.8 Trace Related Commands

1. `^showTrace(s,m,t) ⇒ #tracer:` shows the time-stamped record(s) of commands `m` send by `s` to `t` that are archived at `#tracer`.

2. `^removeTrace(s,m,t) ⇒ #tracer:` removes the record(s) of commands `m` send by `s` to `t` that are archived at `#tracer`.

# Appendix D

# Glossary of Eiffel Interactions and the Rules Controlling Them

For each interaction $\mathcal{I}$ we have a `can_`$\mathcal{I}$ and a `cannot_`$\mathcal{I}$ rule. A `can_`$\mathcal{I}$ rule defines a permission for $\mathcal{I}$ interactions where as a `cannot_`$\mathcal{I}$ rule defines a prohibition for the same. Here we just present all such controllable $\mathcal{I}$s. In the following `C, D,C1` or `C2` stand for program-modules representing eiffel classes.

1. `inherit(C,D)`: class represented by `C` inherits from class represented by `D`.
2. `redefine(C,F,D)`: class represented by `C` redefines feature `F` inherited from class represented by `D`.
3. `rename(C,F,D)`: class represented by `C` renames feature `F` inherited from class represented by `D`.
4. `undefine(C,F,D)`: class represented by `C` undefines feature `F` inherited from class represented by `D`.
5. `call(F1,C1,F2,C2)`: inside feature `F1` of class represented by `C1` a call via feature `F2` is made to class represented by`C2`. `C2` represents the class where `F2` is either defined or last redefined or last renamed as `F1`.
6. `assign(F1,C1,F2,C2)`: feature `F1` of class represented by `C1` makes an assignment to `F2` which is defined in class represented by `C2`.
7. `reverseAssign(F1,C1,F2,C2)`: feature `F1` of class represented by `C1` makes a reverse assignment attempt to `F2` which is defined in class represented by `C2`.
8. `generate(F1,C1,F2,C2)`:feature `F1` of class represented by `C1` generates a new instance of the class represented by `C2` named into `F2`.
9. `use(C,D)`: class represented by `C` is using class represented by `D` as a client.
10. `useC(C,F)`: class represented by `C` is using a C routine `F`.
11. `changeExpStatus(C,F,D)`: class represented by `C` changes the export status of feature `F` that it inherits from the class represented by `D`.

# Appendix E
# List of Built-in MetaRules

1. `canDoMr:` progenitor of all `canDo` rules.

2. `cannot_`$\mathcal{I}$: for each $\mathcal{I}$ there is one such metarule that acts as the progenitor of prohibition rules controlling the interaction $\mathcal{I}$.

3. `can_`$\mathcal{I}$: similarly for each $\mathcal{I}$ there is a metarule that acts as the progenitor of permission rules controlling the interaction $\mathcal{I}$.

4. `simpleCond:` progeny of all auxiliary rules defined by users.

## E.1 List of Built-in Rules

1. heirOf(c1,c2)

2. isIn(a,L)

3. creationOp(M,O)

4. modifierOp(M)

5. observerOp(M)

6. traceOp(M)

7. configurationOp(M)

8. eiffelOp(M,O)

9. legislativeOp(M,O)

# References


[1] Chowdhury Anir and Scott Meyers. Facilitating software maintenance by automated detection of constraint violation. In *IEEE conference on Software Maintenance*, September 1993.

[2] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered sde. *Software Engineering Notes, Special Issue on Fifth ACM SIGSOFT Symposium on Software Development Environments*, 17(5):21–31, December 1992.

[3] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78, March 1992.

[4] N. Belkhtair, J. Estublier, and W. Melo. Adele-tempo: An environment for process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*. John Wiley and Sons, 1994.

[5] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993. (CUCS-012-92, April 1992).

[6] G. Birtwistle, O. Dahl, B. Myhrtag, and K. Mygaard. *Simula Begin*. Auerbach Press, 1973.

[7] Frederick P. Jr Brooks. No silver bullet – the essence and accidents of software engineering. *IEEE computer*, pages 10–19, April 1987.

[8] J. Chomicki and N.H. Minsky. Towards a programming environment for large prolog programs. In *Proceedings of the 2nd International Symposium on Logic Programming*, pages 230–241, Boston, Massachusetts, July 1985.

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

[10] P. Coad. Object-Oriented Patterns. *Communications of The ACM*, 1992.

[11] F. DeRemer and H. H. Kron. Programming-in-the-large vs Programming in the small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[12] T. Despeyroux. Executable specifaction of static semantics. In *Semantics of Data Types, Lecture Notes in Computer Science*, volume 173. Springer-Verlag, June 1984.

[13] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A Metalanguage for C++. In *USENIX C++ Conference*, August 1992.

[14] Gamma E., Helm R., Johnson R., and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of ECOOP-93*, 1993.

[15] S. I. Feldman. Make - a program for maintaining computer programs. *Software Practice and Experience*, 9, April 1979.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Michroarchitectures for Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEE Software*, November 1995.

[18] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific, 1993.

[19] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[20] A. N. Haberman and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12), December 1986.

[21] Y. Huang and C. Kintala. Software implemented fault-tolerance: Technologies and experience. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 2–9. IEEE CS Press, 1993.

[22] W.L. H'ursch, L.M Seiter, and C. Xiao. In any case: Demeter. *The American Programmer*, September 1991.

[23] G. Junkermann, B. Peuschel, W. Schfaer, and S. Wolf. Merlin: Software development through a knowledge-based environment. In *Software Process Modeling and technology*, pages 103–130. Research Studies Press. John Wiley and Sons, England, 1994.

[24] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences* , number 2, pages 131–140, January 1990.

[25] G. Kiczales, J.D. Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, 1991.

[26] B. Krishnamurthy and Naser S. Barghouti. Provence: A Process Visualization and Enactment Environment. In *Fourth European Software Engineering Conference (ESEC 93), Lecture Notes in Computer Science, No. 717*, number 717. Springer-Verlag, September 1993.

[27] B. Krishnamurthy and D. S. Rosenblum. Yeast: A general purpose event-action system. 1993.

[28] ClearCase Concepts. *Atria Software Inc., Natic, Massachussets*, 1993.

[29] D. Leblang and R Chase. Computer aided software engineering in a distributed workstation environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering symposium on practical software development environments*, April 1984.

[30] D. Leblang and R Chase. Configuration management for large scale software development efforts. In *Workshop on Software Engineering Environments for Programming in the Large, Harwichport, Massachussets*, June 1985.

[31] K. Lieberherr and I. Holland. Formulations and benifits of the Law of Demeter. *SIGPLAN NOTICES*, 24(3):67–78, March 1989.

[32] K. Lieberherr and I. Holland. Preventive maintenance of object-oriented software. *?*, 1992.

[33] K. Lieberherr, I. Holland, and A. Riel. Object oriented programming an objective sense of style. *SIGPLAN NOTICES :special issue on OOPSLA 88*, 23(11):323–334, November 1988.

[34] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[35] S. B. Lippman. *C++ Primer.* Addison-Wesley, 1990.

[36] D Luckham, F. von Henke, B. Krief-Bruckner, and O. Owe. Anna, a Language for Annotating Ada Programs: Reference Manual. In *Lecture Notes in Computer Science*, volume 260. Springer-Verlag, 1987.

[37] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[38] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. Technical report, April 1993. Presented in Workshop on Principles and Practice of Constraint Programming.

[39] N. H. Minsky and A. D. Lockman. Ensuring integrity by adding obligations to privileges. In *Procedings from the 8th International Conference on Software Engineering*, 1985.

[40] Naftaly H. Minsky. Independent on-line monitoring of evolving systems. In *International Conference on Software Engineering*, March 1996.

[41] N.H. Minsky. Law-governed systems. Technical Report LCSR-TR-101, Department of Computer Science, Rutgers University, February 1987.

[42] N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.

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

[44] N.H. Minsky. Law-governed regularities in object systems; part 1: Principles. Technical report, Rutgers University, LCSR, December 1994. (Accepted for publication in Theory and Practice of Object Systems (TAPOS)).

[45] N.H. Minsky. Regularities in software systems. In D. Lamb, editor, *Studies of Software Design.* Springer-Verlag, 1994. (To be published).

[46] N.H. Minsky and A. Borgida. The darwin programming environment. Technical Report LCSR-TR-54, Rutgers University, November 1984.

[47] N.H. Minsky and A. Borgida. The darwin software-evolution environment. In *Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments*, pages 89–95, April 1984.

[48] N.H. Minsky and P Pal. Establishing regularity in object-oriented (eiffel) systems. Technical Report LCSR-TR-227, Rutgers University, LCSR, June 1994. (Presented at the ECOOP Workshop on Patterns on OO programming,Bologna, July 1994).

[49] N.H. Minsky and P Pal. Law-governed regularities in object systems; part 2: the eiffel case. Technical Report LCSR-TR-228, Rutgers University, LCSR, December 1994. (Accepted for publication in Theory and Practice of Object Systems (TAPOS)).

[50] N.H. Minsky and D. Rozenshtein. Law-governed systems: Research perspective. Technical Report CAIP-TR-030, CAIP Center, Rutgers University, December 1986.

[51] N.H. Minsky and D. Rozenshtein. Law-based approach to object-oriented programming. In *Proceedings of OOPSLA' 87*, pages 482–493, 1987.

[52] N.H. Minsky and D. Rozenshtein. Controllable delegation: An exercise in law-governed systems. In *OOPSLA' 89*, pages 371–380, 1989.

[53] N.H. Minsky and D. Rozenshtein. Configuration management by consensus: An application of law governed systems. In *ACM Symposium on Software Development Environments*, 1990.

[54] G.C Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high level models. In *Proceedings of the 3rd ACM Symposium on Foundations of Software Engineering*, pages 18–28, Washington, D.C., October 1995.

[55] H. L. Ossher. Grids: A newprogram structuring mechanism based on layered graphs. In *Proceedings of the ACM POPL*, January 1984.

[56] H. L. Ossher. A mechsnism for specifying the structure of large layered systems. In *Research Directions in Object-Oriented Programing*. MIT Press, 1987.

[57] P. P Pal. Towards compile time enforcement in lgs. *Masters Essay submitted to the Department of Computer Science, Rutgers University, New Brunswick, New Jersy*, 1990.

[58] D.E. Perry. The Inscape environment+. In *Proceedings of the 11th International Conference on Software Engineering (ICSE)*, May 1989.

[59] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[60] M. Rochkind. The source-code control system. *IEEE Transactions on Software Engineering*, SE-1:364–370, December 1975.

[61] D. Rozenshtein and N.H. Minsky. Constraining interactions between objects in the presence of class inheritance. In *Proceedings of the 2nd International Workshop on Computer-Aided Software Engineering*, July 1988.

[62] D. Rozenshtein and N.H. Minsky. Law-governed object-oriented system. *Journal of Object-Oriented Programming*, 1(6):14–29, March/April 1989.

[63] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, March 1993.

[64] Beth A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, June 1995.

[65] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996.

[66] M. Shaw. Procedure calls are the assemply language of software interconnection. In *Proceedings of Workshop on Studies of Software Design*, May 1993.

[67] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA'86 Conference*, pages 38–45, September-October 1986.

[68] B. Stroustrap. *The C++ Programming Language.* Addision-Wesly, 1994.

[69] The Ada 9X Mapping/Revision Team. *The Ada 9X Reference Manual (ISO/IEC CD 8652/ISO/IEC JTC1/SC22 N 1451 and ISO/IEC JTC1/SC22 WG9 N 191).* Intermetics Inc, 1994.

[70] W. Teitelman. A tour through Cedar. *IEEE Software*, 1, April 1984.

[71] W. Tichy. Design, implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering (ICSE)*, September 1982.

[72] A. L. Wolf, L. A. Clarke, and J.C. Wileden. Interface control and incremental development in the PIC environment. In *Proceedings of the 8th International Conference on Software Engineering (ICSE)*, August 1985.

# Vita

## Partha Pratim Pal

**1966** Born in Srirampur, West Bengal, India.

**1975** Primary Examination, West Bengal Board of Primary Education.

**1977** Middle English Scholarship, West Bengal Board of Secondary Education, Ranked First in District.

**1982** Secondary Examination, West Bengal Board of Secondary Education, First Division.

**1984** Higher Secondary Examination, West Bengal Council of Higher Secondary Education, First Division.

**1988** Bachelor of Computer Science And Engineering, Jadavpur University, Calcutta, India, First Class Honors, Ranked 7th in the graduating BCSE class.

**1988-1996** Graduate work in Computer Science, Rutgers University, New Jersey.

**1991** M.S. in Computer Science, Rutgers University, New Jersey.

**1996** Ph.D. in Computer Science, Rutgers University, New Jersey.