SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **00**(S1), 1–5 (1999) *Prepared using* speauth.cls *[Version: 1999/06/11 v1.1a]*

Providing Multiple Views for Objects

Naftaly H. Minsky[1], Partha pratim Pal[2*]
[1] *Department of Computer Science*
Rutgers University
New Brunswick, NJ 08903 USA
(email: minsky@cs.rutgers.edu)
[2] *BBN Technologies*
10 Moulton Street, MS 6/3D
Cambridge, MA 02138 USA
(email: ppal@bbn.com)

SUMMARY

The need for multiple views of an object is often encountered in software practice. This paper presents our experience in addressing this need under a software architecture known as the *law-governed architecture*. We introduce the notion of a *surrogate object* which allows an object to appear different and behave differently when used from different parts of a system. This concept requires some minor modifications to the classical inheritance-based object-oriented systems, mostly involving a judicious use of *delegation*. A concrete implementation of surrogates under the law-governed architecture is described and some applications of surrogates are briefly discussed.

KEY WORDS: Object-Oriented Modeling and Frameworks, Surrogate Objects, Delegation, Design Patterns, Law-Governed Architecture, Access Control

Introduction

One often needs to provide different parts of a large and evolving system with different views of a single object [5]. Such views may differ from the base object in question, and from each other, in two ways: (a) they may allow different access to the *intrinsic* features[†] of the base object, and (b) they may associate with the base object some additional, *extrinsic* features which may be meaningful only with respect to a given view[‡].

Consider, for example, a large evolving medical information system $\mathcal{M}$. Suppose that this system is partitioned into several disjoint *divisions*, which are to be developed

*Correspondence to: Partha Pal, BBN Technologies, 10 Moulton Street, MS 6/3D, Cambridge, MA 02138 USA
Contract/grant sponsor: In part NSF and ARPA; Contract/grant number: CCR-9308773 and DABT63-93-C-0064 respectively.
[†]The term "feature" is used in this paper for either an attribute (instance variable) of an object, or for its method.
[‡]The terms *intrinsic* and *extrinsic* for the features of a surrogate object are borrowed from Harrison and Ossher [5].

CCC 0038–0644/99/S10001–05$17.50 *Received 5 May 1999*
 Revised 25 December 1999

more or less independently of each other. These divisions include the `patient-record` division, which maintains a database of `patient` objects; the `medical` division, which deals with the medical status of patients, and with their treatment; and the `financial` division, which deals with fees, insurance and related matters. Both the medical and the financial divisions of the system need to access `patient` objects, but they are likely to require, and be allowed, different views of them. For example, the medical division needs to update the health status of patients, which the financial division should not be allowed to do; and vice versa for the financial status of patients. Moreover, each of these divisions may need to associate certain information with every patient, which is of no concern to the other division, and which may not be considered an intrinsic aspect of the patient itself. For example, the medical division may need to associate with each patient the schedule of his future treatments.

In general, one would like to have a convenient mechanism for constructing such different views for any given class of objects, and for specifying which part, or division, of a system should be presented with which view. Moreover, one would like such views to be maintainable with minimal effort, while the system-divisions that use them evolve. Unfortunately, the classical OO model of programming does not support multiple views of this kind. Harrison and Ossher [5], in particular, state that:

> "designers of such suites ["divisions," in our terminology] are forced either to forego advantages of the object oriented style [such as encapsulation and polymorphism] or to anticipate all future applications, treating all extrinsic information as though it were intrinsic to the object's nature."

Their solution to this problem, called *subject-oriented programming*, involves a fairly radical departure from the classical OO style of programming, which distributes the definition of a class of objects among the various divisions (subjects), each defining its own "subjective view" of it, tying them all together by a globally unique object-identity, and by means of global *composition rules*.

Another related approach is known as *Aspect Oriented Programming* [8], where a software program is divided into different aspects of concern, each of which are designed and implemented separately in a language that is most appropriate for expressing that aspect. The final application is then constructed by *weaving* the different aspects by means of a code generation tool.

In this paper another approach to multiplicity of views, which is more consistent with the classical OO model of programming, is introduced. Under this approach, which is based on the concept of *law-governed architecture* (LGA) [12],[13], views are provided by what we call *surrogate* objects, which *delegate* selected messages to their base objects, but can also act on their own. The relationship between such surrogates and their base objects, and the scope of both kinds of objects, is defined by the *law* of the system, which is enforced by the environment in which the system is developed. It is to be noted that LGA is a general mechanism under which various useful structures, constraints and protocols can be defined as laws and enforced upon software systems, as evidenced in a series of earlier papers [16], [14], [11]. The topic of this paper, defining and imposing surrogate relationship as a means to provide multiple views of objects, is a specific usage of the general LGA mechanism. This paper describes a specific implementation of surrogates using the Darwin-E software development environment [15], which establishes LGA over Eiffel[10] systems. But, in principle, this approach is applicable to any class-based object-oriented language.

The Concept of a Surrogate

Let b⋆ be a class[§] whose instances are to look and behave differently in different parts of a given system. Class b⋆ is called, in this context, the *base class* and its instances are called the *base objects*. Let b be one such base object. A surrogate object s for b is an instance of a class s⋆ that is designed to be a *surrogate class* for b⋆. The relationship between the base object and its surrogates, and the scope of such objects, are to satisfy the following properties.

Property 1. *Every surrogate object* s *has a reference, named* baseRef*, to its base object* b*. A surrogate object provides its clients with two different types of features: the* intrinsic features*, which are a subset of the features of the base object* b*, designated (in a manner specified later) to be accessible through* s*; and the* extrinsic features*, which are the features defined in class* s⋆ *itself. The distinction between these two types of features is* **transparent** *to the clients of* s*.*

Property 2. *A feature* i *of an object* b *designated as an intrinsic feature of a surrogate* s *of* b*, is made accessible through* s **by delegation**, *as follows: every expression* s.i(..) *is transformed automatically into the expression* s.baseRef.i(...)*, which effectively invokes* i(..) *on the base object of* s*. Note that this delegation mechanism is applied uniformly to all surrogates, and does not require any code inside the surrogate classes themselves.*

Property 3. *Any class can serve as a base class, and it can have any number of independently defined surrogate classes. Moreover, any instance of a base class can have any number of surrogate objects, belonging to one or more surrogate classes.*

Property 4. *The specification of which features of a base class are to be made accessible via which of its surrogate classes and in which context, is made by* global rules*, called the* **surrogate rules**.

To illustrate this concept of surrogates, consider a system $\mathcal{S}$ depicted in Figure 8. This system contains, in particular, two divisions, called d1 and d2, each of which consists of any number of classes, which may change during the evolution of the system. Now, consider an object b of class b⋆, which has two surrogates: s1 of class s1⋆, and s2 of another surrogate class s2⋆. The following is assumed about these classes, and about the relation between them:

1. Class b⋆ has the features i1,i2,i3.
2. Features i1,i2 of class b⋆ are the intrinsic features made visible via surrogate class s1⋆. In addition, class s1⋆ defines its own, extrinsic, features e1, e2.
3. Features i2,i3 of class b⋆ are the intrinsic features made visible via surrogate class s2⋆. In addition, class s2⋆ defines its own, extrinsic, features f1,f2.
4. The instances of classes s1⋆ and s2⋆ can be used only in divisions d1 and d2 respectively, as illustrated in the Figure 8. Neither of these divisions is allowed to use objects of the base class b⋆ itself.

Under these assumptions, expression s1.e1, anywhere in division d1, would invoke feature e1 of s1 itself, while the expression s1.i1 would be delegated to the base object b invoking method i1 in it. Note that feature i3 of b is not accessible through

[§] Throughout this paper we denote class names with a star, as in b⋆.

`s1`, although it is accessible through the other surrogate, `s2`. It should be noted that although controlling access to *intrinsic* features (like `i1`, `i3`) is a core aspect of the surrogate relationship between `s1` and `b`, it also entails adding *extrinsic* features (such as `e1`) to the view of `b` provided to the division `d1` by means of `s1`. Later on we show how this can be realized in our environment. The merits of our approach over the obvious *wrapper* or the *adapter design pattern* approach can be found later in the paper.

We conclude this Section with the following observations: First, note that no inheritance relationship is assumed between a surrogate class `s`$\star$ and its base class `b`$\star$. This is because `s`$\star$ may hide some of the features of `b`$\star$, and thus be ineligible as a subclass of `b`$\star$, from the viewpoint of strong typing. Second, note that delegation is not new here; it has been used, in particular, as the basis for the *prototype-and-delegation* style of OO programming [9] — an alternative to class-based programming. There is, however, a subtle difference[¶] between our delegation mechanism and the traditional one. Traditionally, the value of pseudo-variable 'self' (`current` in Eiffel and `this` in C++) is not changed when delegating a message. Preservation of "self" in delegation has been discussed quite extensively in literature since the early days of OO [20], [19], [25], and it is perhaps worthwhile to explain the our position on this matter. In the traditional case of *prototype-and-delegation*, when a receiver `r` delegates a message to a delegatee `d`, the receiver `r` is the *principal*—`d` just acts on behalf of `r`, and remains essentially behind the scene. In our case, on the other hand, the base object and its surrogates are all autonomous objects, operating on their own in their respective divisions, and thus need to retain their own *selfs*. As an example for the need of the base object to retain its autonomy, consider the possibility for it to maintain a log of all its interactions with its surrogates.

An Overview of Law-Governed Architecture

In this section we briefly introduce the concept of law-governed architecture (LGA) and the Darwin-E environment that supports it for Eiffel systems. We will confine ourselves mostly to those aspects of Darwin-E that are directly relevant for our surrogates framework. For more about this environment the reader is referred to [15].

The main novelty of LGA is that it associates with every software development project $\mathcal{P}$ an *explicit* set of rules $\mathcal{L}$ called the *law* of the project, which is strictly *enforced* by the environment that manages this project.

The state of the project under Darwin-E is represented by an object base $\mathcal{B}$. It is a collection of objects of various kinds: including *program-modules*, which, in the case of Darwin-E represent classes, and *builders*, which serve as loci of activity for the people (such as programmers and managers) that participates in the process of software development. The objects in $\mathcal{B}$ may have various *properties* associated with them, which are used to characterize objects in various ways. Syntactically, a property of an object may be an arbitrary prolog-like term, but we use here only very simple cases of such terms whose structure will be evident from our examples. Some of the properties of objects are built-in, that is, they are mandated by the environment itself, and have predefined semantics; others are mandated by the law of a given project, which defines their semantics for the particular projects. As an example of a built

[¶]Some may object to our use of the term *delegation* because of this difference.

in property, a class object `c⋆` that inherit from a class `c1⋆` would have the property `inherits(c1⋆)` associated with it in $\mathcal{B}$. To illustrate the nature and use of properties that may be mandated by the law for a specific project, let us explain how the notion of *divisions*, introduced earlier, can be implemented in darwin-E. This is done, first, by associating the property `division(d)` with any class that we wish to include in this division, and by making the law of the project sensitive to this property of classes. Note that the law of a given project may be written to allow for a single class to belong to several division, if this is desired.

Broadly speaking, the law $\mathcal{L}$ of a given project $\mathcal{P}$ is a set of rules about certain *regulated interactions* between the objects constituting this project. Two kinds of such interactions are recognized:

1. Developmental operations, generally carried out by people, i.e., the builders of the project. These interactions include the creation, destruction and modifications of class-objects, and changes of the law itself by the addition and deletion of rules.
2. Interactions between the component-parts of the system being developed.

The rules that regulate the former kind of interactions, thus governing the process of evolution of $\mathcal{P}$, are enforced dynamically, when the regulated operations are invoked. The structure of these rules has been described in [12], and its knowledge will not be required for the rest of this paper.

The rules that regulate the latter kind of interactions, thus governing the structure of any system developed under $\mathcal{P}$, are *enforced statically* — when the individual class-objects are created and modified and when a system of classes is put together into a *configuration*, to be compiled into a single executable code. The nature of a special case of this second kind of rules, and the type of interactions (henceforth by interaction we will mean interactions of the second kind only) regulated by them, are discussed below. An example of the kind of interactions between the component parts of an Eiffel system that can be regulated under Darwin-E is the relation `call(r, c1⋆,f, c2⋆)` which means that routine `r` of class `c1⋆` contains a call to feature `f` of class `c2⋆`.

There are quite a number of additional interactions that can be regulated by the law under Darwin-E, most of which are discussed in [15]. Each such interaction is regulated by rules specific to it. For example, the call interactions can be controlled by the *cannot_call* and/or the *can_call* rules. As the names suggest, the first kind of rules are used to specify the calls that are to be prevented, while the later are used to specify which calls are admitted, with possible modifications. Figure 1 below contains an example of the *cannot_call* rules[‖]:

$\mathcal{R}1$.
```
cannot_call(_,C1,modify_name,p⋆) :-
        division(medical)@C1| division(financial)@C1.
```
This rule prevents all calls to `p⋆` *by the feature* `modify_name` *from the medical and financial division, with the obvious intention that these classes should not be able to change the name of instances* `p⋆`.

Figure 1. Example of a rule preventing call interaction

[‖] As can be seen in rule $\mathcal{R}1$, we represent a rule as a Prolog clause. The term of the form `p@o` succeeds if the property `p` is present in the object `o`. In the rules upper case letters mean a variable in Prolog sense, whenever we talk about a class `C` or a feature `F`, we actually mean any class that can unify with `C` or any feature that can unify with `F`.

The expressive power of this rule, as compared to Eiffel's selective export, is to be noted: there is no way in Eiffel to define the class `p⋆` in such a way that any class that could ever be in the medical or financial division is denied to use this feature.

Finally, it should be pointed out that the critical process of changing the law itself, by the creation and destruction of rules, is also tightly regulated under LGA, as described in [12].

Implementing Surrogates in Darwin-E

This Section describes how the concept of surrogates is implemented under the Darwin-E environment for programs written in Eiffel. The discussion is kept at a fairly abstract level, postponing the implementation details for a later Section.

The surrogate relation between classes is specified in the law by what we call *surrogate* rules. Such rules have the form:

```
surrogate(C,S,B,F) :- cond(C,S,B,F).
```

where `cond(C,S,B,F)` specifies a condition under which class `S` is to be used as a surrogate for class `B` in the context of class `C`, delegating features `F` to it. The condition `cond(C,S,B,F)` serves another purpose: calls made to a class `S` from a class `C` are always admitted if the condition `cond(C,S,_,_)` is satisfied. In other words, features of a class `S` can be called in the context in which it is designated to act as a surrogate. Therefore, in addition to designating `S` to be a surrogate class that delegates certain intrinsic features in a certain context, a surrogate rule also defines the scope of the surrogates of class `S`, where its extrinsic features can be used.

As an example of the surrogate rules, consider the two rules in Figure 2 that establish the surrogate relations described in Figure 8.

$\mathcal{R}2$. `surrogate(C, s1⋆, b⋆,F) :-`
`division(d1)@C,`
`(F= i1| F= i2).`
Class `s1⋆` *is a surrogate class for* `b⋆`*, in the context of all classes in division* `d1`*, delegating features* `i1` *and* `i2` *to it.*

$\mathcal{R}3$. `surrogate(C, s2⋆, b⋆,F) :-`
`division(d2)@C,`
`(F= i2| F= i3).`
Class `s2⋆` *is a surrogate class for class* `b⋆`*, in the context of all classes in division* `d2`*, delegating features* `i2` *and* `i3` *to it.*

Figure 2. Specification of the surrogate relations described in Figure 8

Rule $\mathcal{R}2$, in particular, makes `s1⋆` a surrogate class of `b⋆`, designating `i1` and `i2` as the intrinsic features accessible through it in division `d1`, and in addition, ensures that surrogates of class `s1⋆` can only be used in division `d1`.

To explain the effect that surrogate rules have on the system developed under Darwin-E, consider some class `s⋆` that according to such rules serves as a surrogate for class `b⋆`, and let `i` be one of the features delegated from `s⋆` to `b⋆` in the context of all classes in a division `d`. Let class `c1⋆` be a class in division `d` and class `c2⋆` be a class that is not in division `d`. Then:

1. Class `s⋆` is forced to have a universally exported** attribute named `baseRef`.
2. Every expression `s.i(..)` in class `c1⋆`, where `s` is a variable of class `s⋆` is transformed into `s.baseRef.i(..)`, thus effecting the required delegation.
3. Features of class `s⋆` (i.e. extrinsic features of the surrogate) can be called from class `c1⋆` because such calls will be admitted by darwin-E, whereas such calls from `c2⋆` will not be admitted. Therefore, class `c2⋆` will not be able to use instances of class `s⋆`.

Limiting the Scope of Base Classes

In the last Section we saw that the surrogate rules establish the scope of surrogate classes, for a given base class `b⋆`. But what about the scope of the base class `b⋆` itself? When presenting a given part of a system with a certain view of class `b⋆` we do not want it to have direct access to `b⋆` itself. Fortunately, Darwin-E provides ample means for controlling the usage of arbitrary classes. In particular, the ability to call objects of a given class is controllable by the *cannot_call* rules mentioned earlier. For example, Rule $\mathcal{R}4$ in Figure 3 prevents divisions `d1` and `d2`, of Figure 8, from being able to call instances of the base class `b⋆` directly. This is in spite of the fact that every surrogate

$\mathcal{R}4$. `cannot_call(_,C1, _,b⋆) :-`
`division(d1)@C1|division(d2)@C1.`
Classes in divisions `d1` *and* `d2` *cannot call objects of the base class* `b⋆`.

Figure 3. Specifying the Scope of a Base Class

object, such as `s1` in Figure 8, has a public attribute `baseRef` that provides a pointer to the base object of `s1`. Due to Rule $\mathcal{R}4$ above, this pointer cannot be used in division `d1` to call any of the features provided by the base object.

The Creation of Surrogate Objects

Consider a class `s⋆` that serves as a surrogate for a base class `b⋆` in a certain division `d` of a large system. Let us assume that the base objects are not available in division `d`, so that `d` can operate on a given instance `b` of `b⋆` only through a surrogate. But how does it get such a surrogate? There are two basic ways for doing that. First, `d` may be given ready made surrogate objects. It is quite reasonable to assume that there is a specific division, such as the `patient-record` division of the medical information system $\mathcal{M}$ discussed at the beginning of the paper, to manage the creation of base objects as well as the various surrogates, and to distribute these surrogates to their respective clients. The second way is to create a surrogate object with void `baseRef` attribute inside division `d` and pass it to an instance of a class to which base objects are visible, via a function that returns the surrogate after installing the appropriate base object in it. Such a class in our example medical information system $\mathcal{M}$ may as well be a part the `patient-record` division. Note that in either case, the `patient-record` division may not be able to use patient objects because of a `cannot_call` rule like the

**Attribute `baseRef` needs to be universally exported for the transformation defined in (2) above to be meaningful, but, this does not necessarily mean that attribute `baseRef` is usable for anything but the delegation described here. Refer to Section for more on this.

ones discussed in the previous section, however such a rule does not prohibit storing or passing of such objects in this division.

Storage of Surrogate Objects

The dissociation of view specific extrinsic features from the base object by means of the surrogates raises a natural question. The surrogates may contain extrinsic data that logically belong to the base object, and the base object may contain its intrinsic data: what happens when such data need to be stored in persistent storage such as a database? The answer is really simple because the base and the surrogate objects are independent and autonomous. When required, each division will simply store the object that is in its view. For instance, in the medical information system mentioned earlier, the `financial` or `medical` division will store a surrogate object through which it accesses a patient. This means that a surrogate class will provide an operation that stores the extrinisic data of the patient that exists in the view of that division. In addition, the division can make use of an intrinsic store operation which is in its view. Logically, all data (intrinsic and extrinsic) associated with a base object can be thought of as a row in a relational table: some of the columns in this row are filled by the intrinsic data from the base object and rest of the columns are filled by the extrinsic data from the surrogate objects. How the different store operations mentioned above fill in the different cells of the row is an implementation issue, which depends on the underlying database systems.

Inheritance of the Surrogate Status

One may want the surrogate status of a class `s`$\star$ to be inherited by its child class `s1`$\star$. That is, if `s`$\star$ acts as a surrogate of `b`$\star$, delegating features `i1,i2,..` in some context, one may like `s1`$\star$ to serve as a surrogate class for `b`$\star$ in the same context, delegating `i1,i2,..` appropriately. Our model does not provide for such inheritance of the surrogate status, but it can be easily arranged by means of appropriate surrogate rules. In particular, for the case of single inheritance this is provided by means of the following rule:

$\mathcal{R}5$.
```
surrogate(C,V,B,F) :-
        heirOf(V,V1), surrogate(C,V1,B,F).
```

The goal `heirOf(c1,c2)` is resolved by a built-in darwin-E rule, it succeeds if `c1` is a descendant of `c2`. Note that the case of multiple inheritance is more problematic, and will require more complex surrogate rules, which we will not discuss here.

Control Over The Creation and Destruction of Rules

As has already been pointed out, Darwin-E provides means for controlling the evolution of the law by the creation of new rules, and by the destruction of rules. This control is beyond the scope of this paper, but in order to illustrate what it can accomplish we will give just one informal example. It is possible to initialize a software project under which the rules that specify the surrogate relation with respect to a given base class `b`$\star$, and the rules that specify the visibility of `b`$\star$ and of all its surrogate classes, can be created (and destroyed) only by the programmer that "owns" `b`$\star$. (The

concept of ownership, and its implications, can also be defined precisely by the law of a project.)

Other Miscellaneous Issues

Surrogates under the LGA is a novel programming construct. So far we have described the need for surrogates, their implementation details and the core issues involving the surrogates concept. But a practitioner might face a different set of issues while trying to design a system using surrogates. In this Section we address a few of such practical issues with the hope that this discussion provides a general guideline.

Features of a Surrogate Class: Delegate or Define as Extrinsic?

Perhaps the most common dilemma a practitioner will face is in deciding whether a feature of a surrogate class should be *intrinsic* or *extrinsic*. The guiding principle in this matter is the fact that our code-transformation based delegation can dispatch only to the base class implementation (of the intrinsic feature). So, if the feature implemented in the base class provides only a part of the task that the surrogate needs to perform, or the design of the system has an underlying assumption to *wrapper* all access to the base object (for example, counting access or generating notification per access), making the base class feature available as an *intrinsic* feature of the surrogate will not be enough. One has to define appropriately designed extrinsic features that, in addition to dispatch to the proper base class feature, will perform other complementary or *wraperring* actions.

Access to Base Objects: Prevent Altogether or Selectively?

We have discussed earlier that the scope of the base class should be controlled and shown how this can be done using our rules. The decision a practitioner has to take here is how permissive or restrictive he should be in allowing direct access to the base objects. A general guideline is that if a region is supposed to have a certain view by means of surrogates, that region should not have direct access to the base object. We have shown earlier in Rule $\mathcal{R}4$ how to enforce such a constraint by preventing direct access to base objects from divisions `d1` and `d2`. This constraint however, does not prevent access to the base object from other regions in the system. For some application it may be sufficient and for others a more elaborate or strict control may be necessary. Fortunately, LGA provides a powerful and flexible mechanism to accommodate a wide range of logical constraints in such rules. For example, if the system at hand has a requirement to *wrapper* all access to the base class `b`$\star$ then it is important to prevent direct access to base class altogether. This can be achieved by introducing the following rule:

$\mathcal{R}6$. `cannot_call(_,_,_, b*) :- true.`

As another example, a system may require that except for specially designated administrative division(s) no other region should have access to the base object. This requires the following rule to be included in the law of the system:

$\mathcal{R}7$. `cannot_call(_,C,_, b*) :- not division(sysadmin)@C.`

Object Reference: Surrogate or Base?

In some applications it may be necessary to *identify* the object pointed to by an object-reference for various purposes like counting, indexing or locking the object. In the context of surrogates, it must be clearly understood that the object pointed to by the reference to a surrogate object is **not** the base object. Misunderstanding of this fact may result in potentially incorrect results. For instance, locking the object pointed to by a reference to a surrogate object does not lock the base object. So if two surrogate classes make one intrinsic feature available to different regions, then both regions will be able to lock their respective surrogates and invoke the shared intrinsic feature. To ensure mutual exclusivity one has to explicitly *wrapper* the base class feature in the two surrogate classes.

Use of Surrogates

A Paradigmatic Example of the use of Surrogates

Let us reconsider the medical information system $\mathcal{M}$ introduced in the beginning. Suppose that the features of class `patient`$\star$ are categorized into three groups: `personal`, `medical` and `financial`, and let this categorization be defined by means of a property `feature(CATEGORY,F)` associated with the object representing class `patient`$\star$ in the project database. Thus, for example, the property `feature(personal,name)` means that `name` is a `personal` feature of a patient, and a property
`feature(medical,blood_pressure)` means that `blood_pressure` is a `medical` feature of a patient. (Note that this scheme allows for a particular feature `f` to be defined in more than one category.) These categories of features are used in defining the views of the `medical` and `financial` divisions as follows. The `medical` division needs to use the `personal` and `medical` features and the `financial` division needs to use the `personal` and `financial` features of patients. In addition, each division needs to associate a set of *extrinsic* features with their respective view of the patients. The `medical` division is implemented in terms of the surrogate class `pm`$\star$ and the `financial` division in terms of the surrogate class `pf`$\star$ and neither division is allowed to use `patient`$\star$. The class `pm`$\star$ [`pf`$\star$] defines the extrinsic features needed by the `medical` [`financial`] division and delegates the desired intrinsics. All this is established by the rules in Figure 4.

How effective is our notion of surrogates? How easy does it make the independent development of different divisions of the system, which share some common types of objects. How flexible does it make the process of evolution of large systems? These are the questions we address next.

First, note that the base class is completely independent of the nature and the very existence of its surrogate classes, or the different divisions that use these surrogates. One can add a new surrogate class, or change an existing one without any change to the base class. Second, the various surrogate classes are completely independent of each other, unless some dependency is explicitly built into them, as it might, of course, for any collection of classes in an object oriented system. Third, a surrogate class `s`$\star$ may be either completely independent of its base class `b`$\star$, or only partially dependent on it. `s`$\star$ is completely independent of `b`$\star$, if it has no extrinsic features. It then functions as a kind of partially reflective mirror, which is used for access control.

$\mathcal{R}8$. `surrogate(C,sm⋆, patient⋆, F) :-`
`division(medical)@C,`
`(feature(medical,F)@patient⋆ |`
`feature(personal,F)@patient⋆).`

Class `sm⋆` *is a surrogate class for* `patient⋆` *in the context of all classes in* `medical` *division, delegating the* `medical` *and* `personal` *features to it.*

$\mathcal{R}9$. `surrogate(C, pf⋆, patient⋆, F) :-`
`division(financial)@C,`
`(feature(financial,F)@patient⋆ |`
`feature(personal,F)@patient⋆).`

Class `sf⋆` *is a surrogate class for* `patient⋆` *in the context of all classes in* `financial` *division, delegating the* `financial` *and* `personal` *features to it.*

$\mathcal{R}10$. `cannot_call(_,C,_,patient⋆) :-`
`division(medical)@C| division(financial)@C.`

Calls to the base class `patient⋆` *from classes in* `medical` *and* `financial` *divisions are prohibited, therefore such divisions cannot use the base objects.*

Figure 4. Rules for the Medical Information System $\mathcal{M}$

No change is required in such a surrogate when new features are built into the base, or when features are removed from it. The surrogate itself is in fact quite empty in this case. If a surrogate class `s⋆` has features of its own, then the code built into it might access its base object. If the features of the base object accessed by `s⋆` are changed, then, unavoidably, the code of `s⋆` may have to be changed as well. It is possible, however, to restrict a priori such potential dependence of the surrogates on their base, by preventing a surrogate class from calling some, or all, of the features of the base class. For example, we may prevent class `s⋆` from calling *any* of the features of the base by means of the following rule:

$\mathcal{R}11$. `cannot_call(_,s⋆,_,b⋆) :- true.`

Alternatively, we may prevent class `s⋆` from calling the financial feature of its base by means of the following rule:

$\mathcal{R}12$. `cannot_call(_,s⋆,F,b⋆) :- feature(financial, F)@b⋆.`

This way it is possible to control the degree of dependence of a surrogate class on its base class. Next, divisions with different subjective views of a given base class may evolve reasonably independently of each other, and of the base class, because all they have to know about the objects in question is what is presented by their own surrogate class. Finally, the nature of the rules defining the surrogate relation are global, in the sense that the constraints they express are in terms whole groups of classes, and spans over the entire life time of the project. For example, consider a rule that specifies that the intrinsic features `i1,i2,..in` will be delegated from `s⋆` in the context of classes in the `medical` division. Not only does this rule prevent *all* classes currently outside this division from using these intrinsic features via surrogates of class `s⋆`, it also makes sure that no class outside the `medical` division will *ever* be able to do so.

Reusing Legacy Objects by Means of Surrogates

With passage of time, existing object-oriented systems will give rise to a vast number of *legacy* objects: objects that reside in persistent storage, containing important data that needs to be maintained and reused. But legacy objects may not be (re)usable in a new application as originally defined. As an example, assume that there exists a database of objects of class `employee⋆` representing the employees of a given organization. Now, consider a system $\mathcal{S}$, that is being designed for evaluating the employees of this organization under the equal opportunity law. Such a system should treat the employees as nameless, gender-less entities without having any race, color or religion, which means that various features of the `employee⋆` class must be hidden in $\mathcal{S}$. Defining $\mathcal{S}$ in terms of a subclass of `employee⋆` which hides these features will not work, since existing (legacy) objects of class `employee⋆` cannot be passed onto $\mathcal{S}$, because of type rules (subtype polymorphism) of the underlying language.

We propose to deal with this situation by defining $\mathcal{S}$ in terms of an appropriately defined surrogate class `es⋆`. The surrogate rule that establish the desired surrogate relation is described in Figure 5. The cannot_call rule that does not allow the classes in $\mathcal{S}$ to use the bare legacy objects is also shown in Figure 5. Of-course, surrogates

$\mathcal{R}$13. `surrogate(_,es⋆, employee⋆,F) :-`
`F= i1|F= i2|F=i3|...`
Class `es⋆` *is a surrogate class for* `employee⋆` *in the context of any class in system* $\mathcal{S}$*, delegating only the features* `i1,i2,..` *to it.*

$\mathcal{R}$14. `cannot_call(_,_,_,employee⋆) :- true.`
Calls to `employee⋆` *are illegal any where in system* $\mathcal{S}$*, which means instances of this class cannot be used in the system.*

Figure 5. Rules for Reusing Legacy Objects in System $\mathcal{S}$

are by no means, the only solution. Conventional *wrappers*, *adapters*, *handles* or even *smart pointers* can be utilized for this purpose. However, as we pointed out in Section , such approaches are not quite as good as our surrogates.

Using Surrogates For Dynamic Access Control

An important application of surrogate objects is as a means for providing dynamic access control, which is a regime that determines "who can do what to whom." Access control is needed in almost every complex system that allows for the sharing of object between its various parts, and it is used extensively extensively in operating systems and in databases. Yet, access control is generally not supported by programming languages, under which "access to data is [ordinarily] provided on all-or-nothing basis," as stated by Jones and Liskov [6]. We show here that the capability-based access control mechanism [2] can be implemented efficiently, with no run-time overhead, by means of surrogates.

Consider, for example, a network of nodes connected by means of unidirectional conduits called *pipes*, as illustrated in part (a) of Figure 9. When modeling this network in an object-oriented program, let all pipes be instances of a single class `pipe⋆` that provides the methods `push`, `pull` and `inspect`, with the implied semantics; let there

be several different classes of nodes; and let there be some *observer* objects that inspect the status of various pipes.

Every pipe, such as `p` in part (b) of Figure 9, is shared by (accessible to, by pointers) the two nodes it connects, such as `x` and `y` for pipe `p`, and possibly by some observer objects such as `i`. But such sharing of a pipe should not imply equal rights with respect to it. As suggested by part (b) of Figure 9, only node `x` should be able to apply `push` to pipe `p`, only node `y` should be able to apply `pull` to `p`, and observer `i` should be able to apply only `inspect` to this pipe.

$\mathcal{R}$15. `surrogate(_,p1⋆, pipe⋆,F) :- F= push| F=inspect.`
Class `p1⋆` *is a surrogate class for* `pipe⋆` *in the context of any class in the system, delegating* `push` *and* `inspect` *to it.*

$\mathcal{R}$16. `surrogate(_,p2⋆, pipe⋆,F) :- F= pull| F=inspect.`
Class `p2⋆` *is a surrogate class for* `pipe⋆` *in the context of any class in the system, delegating* `pull` *and* `inspect` *to it.*

$\mathcal{R}$17. `surrogate(_, p3⋆, pipe⋆,F) :- F= inspect.`
Class `p3⋆` *is a surrogate class for* `pipe⋆` *in the context of any class in the system, delegating only a single feature* `inspect`*, to it.*

$\mathcal{R}$18. `cannot_call(_,_,_,pipe⋆) :- true.`
Calls to instances of `pipe⋆` *is not allowed from any class in the system.*

Figure 6. Rules for Dynamic Access Control over Pipes

In fact, however, under most conventional languages, any object that has a pointer to pipe `p`, can apply to it any of its method. Object `x`, in particular, can perform the operation `p.pull`, violating the integrity of the network. (We note here that in Eiffel the various features of a class can be exported *selectively* to different parts of a system. But this does not help us here because object `x`, in particular, needs access to both `push` and `pull`, since some of the pipes lead into it and some lead away from it.)

The network in question can be modeled much more faithfully and safely as follows: One defines three surrogate classes for `pipe⋆`, called here `p1⋆`, `p2⋆` and `p3⋆`. These classes, which do not have any method of their own, are governed by the surrogate rules in Figure 6. The cannot_call rule in Figure 6 makes sure that a pipe can only be used by means of its surrogates. Note that there is no constraint about the scope of the different surrogates, since they are used in every part of the system.

Now, let `p1` be an instance of class `p1⋆` that serves as a surrogate for pipe `p`, and analogously for `p2` and `p3`. The situation in part (b) of Figure 9 would be modeled correctly by giving objects `x`, `y` and `i` not `p` itself, but surrogates `p1`, `p2` and `p3` of `p`, respectively.

These surrogate objects serve here in the same role served by *capabilities* in operating systems [2]. The surrogate `p1`, for example, carries the rights to invoke methods `push` and `inspect`, because these are the features which, due to rule $\mathcal{R}$15, would be delegated to the base object `p` of its surrogate. Moreover, the ability to create surrogates can be regulated by the selective export capability, and surrogate, just like capabilities, can be transferred from one place in a system to another.

Finally, let us point out an important difference between the current example and the example using legacy objects. In case of legacy objects, we had only one surrogate class that hides certain features of the base class from all classes in the system. In

this section, on the other hand, we confront the situation where a single object may need different capabilities for different pipes. One surrogate class for the pipes is not sufficient for this purpose, because at best, such a surrogate could delegate `push` in one context and `pull` in another, whereas a single node may need to `push` one pipe and `pull` from another pipe.

Implementation Details: Interpretation and Enforcement of Surrogate Rules Under Darwin-E

This Section explains the following aspects of the implementation of surrogates under Darwin-E:

1. How a surrogate class is forced to define and export its `baseRef` attribute appropriately.
2. How delegation from surrogate to their base object is actually carried out automatically.
3. How the scope of the surrogate objects are established.

For complete understanding of what follows one probably needs to be familiar with our paper about Darwin-E [15]. But the following definitions from that paper will supply some of the needed information.

Definition 1 (call interaction) *The interaction* `call(f1,c1⋆,f2,c2⋆)` *occurs, if the feature* `f2` *which* originates *in class* `c2⋆` *is called from routine* `f1` *of class* `c1⋆`.

A feature `f` *originates* in a class `c⋆` if `c⋆` defines `f` or redefines the inherited feature `f` or renames some other inherited feature as `f`. In case darwin-E finds a client calling a supplier by a feature that does not originate in the supplier class or any of its ancestors, it notes the call as a potential candidate for delegation. For example, a remote call of the form `s.i(..)` in a routine `f1` of class `c⋆` where the feature `i` is not defined in `s⋆` (the class of `s`) or in any of its ancestors will result in the interaction

```
call(f1,c⋆,i,delegate(s⋆)).
```

Note that the above mechanism of detecting which features are to be delegated is specific to Eiffel and Darwin-E. This approach may not always work in languages like Java [1], where it is possible to have interface types (containing method declaration), to be implemented by implementation classes (that contain the method definition). This makes the above definition of *originates* ambiguous. It is possible to address this ambiguity by other means, but that is outside the scope of the present paper.

Definition 2 (include interaction) *Given a class* `c` *and a configuration* `g`, *we say that the interaction* `include(c,g)` *occurs if* `c` *is included in* `g`.

A configuration in Darwin-E represents a collection of classes to be assembled together to form a runnable system.

Now, in order to use our notion of surrogates and delegation in a given project, one must include the rules of Figure 7 in the initial law of that project. With these rules, whenever a class is included in a configuration, the include interaction thus generated invokes rule $\mathcal{R}19$. This rule prevents any class `s⋆` which is designated as a surrogate

```
R19. can_include(S,_) :-
           surrogate(_,S,B,_), surrogate_constraint(S,B).
R20. can_call(F1,C1,F2,delegate(C2)) :-
           lastCalledForm(F)@C1, lastCalledEntity(E)@C1,
           surrogate(C1,C2,B,F2) -> $do(delegate(F1,F,E)@C1) |
           $do(error([F,' not to be delegated'])@C1).
R21. can_call(_,C1,_,C2) :-
           surrogate(C1,C2,_,_).
R22. surrogate_constraint(S,B) :-
           defines(attribute(baseRef),of_type(T))@S,
           exports(baseRef,to([all]))@S, className(T)@B.
```

Figure 7. The Rules For Surrogates and Delegation

of class `b`$\star$ by some surrogate rule, to be included in any configuration if `s`$\star$ and `b`$\star$ do not satisfy the `surrogate_constraint` rule $\mathcal{R}22$.

The `surrogate_constraint` rule approves `s`$\star$ to be a surrogate of `b`$\star$ if `s`$\star$ defines an attribute `baseRef` of class `b`$\star$ and exports it universally. Therefore, if a class is designated as a surrogate class for some base class by some surrogate rules in a project under darwin-E, it could be included in any Eiffel system only if it satisfies the surrogate constraints.

If a class `s1`$\star$ inheriting from a surrogate class for `b`$\star$ is also designated to be a surrogate class for `b`$\star$ then it need not define and export and the `baseRef` attribute of class `b`$\star$ since it already inherits one. But, it should not rename, undefine, change the export status or redefine it. This condition is not easy to express in our rules (although it is possible), specially if `s1`$\star$ is a heir of `s`$\star$ but not a direct child of it. In order to simplify the situation, we require that `s1`$\star$ redefines `baseRef` of its own to be of class `b`$\star$ and export it universally so that `s1`$\star$ satisfies the surrogate constraint defined in rule $\mathcal{R}22$.

Similarly, under the initial law of Figure 7, the call interactions that are potential candidate for delegation are captured by the rule $\mathcal{R}20$. The properties `lastCalledForm(_)` and `lastCalledEntity(_)` are set up by darwin-E storing various information about the call at hand. For example, if `s.i(..)` is the call under consideration then `s` will stored as `lastCalledEntity(s)` and `i(..)` will be stored as `lastCalledForm(i(..))`.

The term `$do(delegation(F1,F,E)@C1)` is a darwin-E code transformation primitive. If the primitive is fired for the interaction `call(f1,c`$\star$`,i,delegate(s`$\star$`))` then `F` is bound to `i(..)`, `F1` is bound to `f`, `C1` is bound to `c`$\star$ and `E` is bound to `s`. As the result of executing the primitive darwin-E will transform the construct `s.i(..)` in the body of routine `f` in class `c`$\star$ to `s.baseRef.i(..)`. According to the rule above, this primitive will only be fired for `call(f1,c`$\star$`,i,delegate(s`$\star$`))` if there is a surrogate rule that designates `s`$\star$ as a surrogate class, delegating the feature `i` to its base class. Therefore, if a feature `f` is declared to be delegatable from a surrogate class `s`$\star$ by some surrogate rule in a project, a call to an instance of `s`$\star$ via `f` will be transformed by darwin-E into a call to its `baseRef`. This transformation is done at compile time when the law is enforced.

The rule $\mathcal{R}21$ approves any call from a class `C1`, to a class `C2` if `C2` is a surrogate class in the context of `C1`, as determined by the goal `surrogate(C1,C2,_,_)` in its body. Therefore, under the initial law of Figure 7, a surrogate rule of the form `surrogate(C,S,B,F):- cond(C,S,B,F)` also defines the context in which the surrogates of class `S` can be used. Unless permission is provided by separate can_call rules, a surrogate is usable only in the context in which it delegates its intrinsic features as defined by the surrogate rules.

Related Work

Wrappers and Similar Devices

Structurally, there is a similarity between our surrogates and the *wrappers* or *adapters*[††] of Gamma et al. [3], the handle-body idiom in C++ [22], the accessors in OATH [7], or the template-hook composition meta pattern of Pree [17]: all of them enclose a *component* much like our *base object*. Of the two design patterns, the adapter pattern is structurally more close to surrogates than the wrapper pattern because unlike the wrapper, the adapter does not need to conform to the component class in much the same way a surrogate class does not need to conform to the base class. It is possible to make the surrogate class conform[‡‡] to its base class, but that would mean that surrogates will not be able to selectively hide the intrinsic features. On the other hand both the wrapper and adapter patterns imply a strong coupling with the component class. The strong coupling arises because of the fact that the wrapper [or adapter] has to accept and redirect the invocation of features defined in the component class, which means that any change (such as adding a new feature) in the component corresponds to a change in the wrapper[adapter]. In our case, no such coding is required for intrinsic features, because the dispatching is done automatically by delegation. This makes our surrogates easier to construct and because we in-line the dispatching of intrinsic feature calls upon a surrogate, it saves a context switch at run time as well.

Visibility of component class features accessible from a wrapper [or adapter] is essentially determined by the visibility of the corresponding dispatching feature of the wrapper [adapter] class. Therefore, any change in visibility warrants code change. For instance, if we do not want to make a feature of the component class accessible through the wrapper anymore, we have to change the code in the wrapper class. In our case only a surrogate rule needs to be changed. In addition, there is a fundamental problem with the conventional approach to visibility control that is relevant here and discussed below.

Let us assume that a feature `i` of the component should be made available only to a division `d` through the adapter. This means that the corresponding dispatching feature in the adapter class is exported only to classes in `d`. First, note that conventional OO languages such as C++, do not provide for this kind of export: one can either make a feature private, protected or public. However, in Eiffel one can use the selective export facility, but this requires that the names of all classes in `d` is stated in the export clause

[††]Gamma et al. use *wrappers* and *adapters* synonymously to denote the same pattern. For simplicity, we will use the term *wrapper* to denote the class version and the term *adapter* to denote the object version of the pattern.

[‡‡]This could be done by forcing `s`$\star$ to a) inherit from `b`$\star$ and b) redefine the inherited features in a desired way by means of darwin-E rules. Such rules are beyond the scope of the present paper.

of feature `i` in the adapter class. Consequently, the code of the adapter needs to be modified every time a new class is added to `d` or an existing class is removed from `d`, in order to satisfy the requirement that `i` is available to division `d`. Such situations are fairly common during the evolution of the system. In our case, we do not need to change the code or the surrogate rules. This is so because our surrogate rules can define the context (i.e. the division `d`) in which `i` is delegated to the base object *by intention*, without actually specifying the members of `d` in the code of the surrogate class.

Finally, the protection offered to the components by the wrappers [or adapters] can be broken easily: any class in the system can declare a variable of the component class and use it by-passing the wrappers [adapters], whereas using the cannot_call rules, we can prevent the use of base objects altogether in a system.

Because of all these reasons, we claim that surrogates are more flexible and more readily reusable than wrappers or adapters. A similar observation can be made about the other devices as well.

Overloading $\rightarrow$ in C++

The forwarding performed by our surrogates can be implemented very easily by overloading the $\rightarrow$ operator of C++ [22] as in *smart pointers* [23]. Therefore C++ specific devices such as the smart pointers or handle-body can perform the kind of delegation we are proposing here, but with certain problems as explained below. The overloading of $\rightarrow$ in C++ is implemented by redefining the $\rightarrow$ operator in the enclosing class (the smart pointer class or the handle class). This means that there is no fine control over which features are delegated under which circumstances: all invocation of features via the overloaded $\rightarrow$ operator will be delegated. In addition, this approach (of overloading the $\rightarrow$ operator) is not safe, because it incorporates the risk of leakage of pointers, a phenomenon similar to exposing the base object via the `baseRef` of our surrogates. While we have the `cannot_call` rules to protect the object attached to `baseRef`, they do not have any such mechanism. The accessors in OATH solves this problem by writing dispatching code as in wrappers, in the accessor class.

Automated Delegation in Jamie

In a recent paper [24], automated delegation by means of code transformation performd by preprocessors is used to enhance Java. Although the objective of this effort to emulate a Java class *extending* (in Java sense) more than one Java classes, this is the only other effort known to the authors that have attempted static analysis and code transformation for automated delegation.

Subject Oriented Programming

Methodologically, our model of surrogates is closely related to that of Harrison and Ossher[5, 4], and to a lesser extent to that of Shilling and Sweeney [18]. Like in the model of Harrison and Ossher, we distinguish between the intrinsic features of an object and the extrinsic features needed by different sets of clients, and we facilitate the independent evolution of different divisions of a system. But there are significant technical differences between these two models. They take a bottom up approach, of constructing an object out of fragments defined in different *subjects*, tying them all

together by a globally unique object-id, and by composition rules. Our approach is essentially top down, from the viewpoint of a given object type. We start with a base object that has some features as its intrinsic part and then provide for different views by creating different surrogates of it. Our approach is simpler, and is squarely within the classical object-oriented paradigm, but it is not as general as theirs from the viewpoint of composing *subjective views* that are implemented in a distributed manner. There is also another difference in the manner in which the relationship between application parts and the corresponding views they use are established. In our case this is defined by surrogate rules that are enforced by a higher level authority (LGA), whereas in SOP, it is limited to language rules such as scope and export. Finally, since SOP maintains a sense of "communal" object identity, it is possible for one subjective view to know about the existence and state of other views in other subjects. Our position, on the other hand, is that the different surrogates of a single base object should be independent of each other's structure and evolution, to allow for incremental evolution of the system. Therefore, although it is possible to give the various surrogates some knowledge of each other, this is not our intention, and is not made automatic.

Aspect Oriented Programming

Our surrogates could be thought of implementing different *aspects* of an object, like under AOP [8]. But our approach is quite different than that of AOP, and has different objectives. AOP and its weaving tools focus on implementations of invidual aspects and composing them into a single system. After such weaving, all aspects of a given object are combined into a single object, with a single identity. This is contrary to the situation here, where the surrogates retian their identities. This makes the evolution of AOP-based system less incremental, and "heavier". Any change in one aspect must be first "weaved" with other aspects before it can be used—as apposed to surrogates, which can evolve quite undependently of each other.

Conclusion

We conclude with an observation about the use of delegation in this paper. Delegation has been considered by some researchers [9] an alternative to classes and inheritance, as an underlying principle in object oriented programming, and there has been some debate about the relative merit of these two principles [21]. What this paper suggests, perhaps, is that the two principles can be usefully combined. Here we have shown that delegation can help in facilitating multiple views. And we believe that, if properly controlled [16], delegation could have some additional applications for class-based languages, which deserve some careful examination.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their insightful comments, which has helped immensly in improvng the quality of the paper.

REFERENCES

1. K. Arnold and J Gosling. *The Java Programming Language.* Java Series. Addison-Wesley, 1998. Second Edition.

2. R.S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Michroarchitectures for Reusable Object-Oriented Software*. Addison-Wesley, 1994.
4. W. Harrison, M. Kaplan, A. Katz, V. Kruskal, E. Lan, and H. L. Ossher. Prototype support for subject-Oriented Programming. In *OOPSLA' 94 Workshop on Subjectivity*, 1994. Position Paper.
5. W. Harrison and H. L. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA'93*, pages 411–428, 1993.
6. A.K. Jones and B.H. Liskov. A language extension mechanism for controlling access to shared data. *IEEE Transactions on Software Engineering*, pages 277–285, december 1976.
7. Brian Kennedy. The features of the object-oriented type hierarchy (OATH). In *Proceedings of the Usenix C++ Conference*, pages 41–50, 1991.
8. G. Kiczales, J. Irwin, J. Lamping, J M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. *ACM Computing Surveys*, 28(4es), December 1996.
9. H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings of the OOPSLA'86 Conference*, pages 214–223, September-October 1986.
10. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
11. N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.
12. N.H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991.
13. N.H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.
14. N.H. Minsky. Regularities in software systems. In D. Lamb, editor, *Studies of Software Design*, Lecture Notes in Computer Science, pages 49–63. Springer-Verlag, 1996. (Number 1078).
15. N.H. Minsky and P. Pal. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(2), 1997.
16. N.H. Minsky and D. Rozenshtein. Controllable delegation: An exercise in law-governed systems. In *Proceedings of the OOPSLA'89 Conference*, pages 371–380, October 1989.
17. Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.
18. J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *OOPSLA' 89*, pages 353–361, 1989.
19. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA'86 Conference*, pages 38–45, September-October 1986.
20. L.A. Stein. Delegation is inheritance. In *Proceedings of the OOPSLA'87 Conference*, pages 138–146, October 1987.
21. L.A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The treaty of orlando. In Won Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Applications and Databases*. Addison-Wesley, 1988.
22. B. Stroustrap. *The C++ Programming Language*. Addision-Wesly, 1994.
23. Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Proceedings of the Usenix C++ Workshop*, pages 1–22, November 1987.
24. J. Viega, Tutt B., and R. Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical report, University of Virginia, Department of Computer Science, March 1999. UVA TR CS-98-03.
25. Mario Walczko. Encapsulation, delegation and inheritance in object-oriented languages. *The IEE Software Engineering Journal*, 7(2):95–101, March 1992.

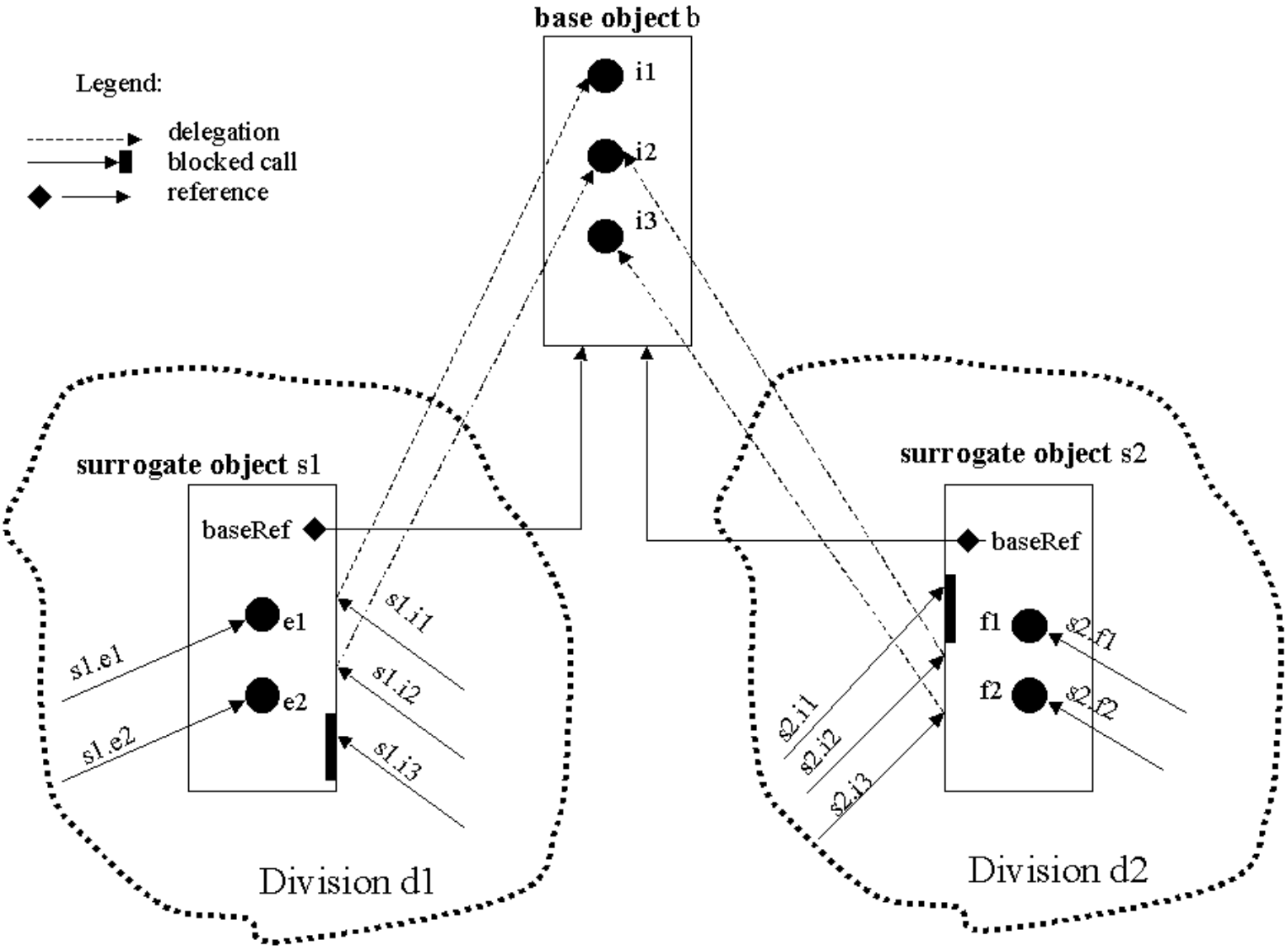
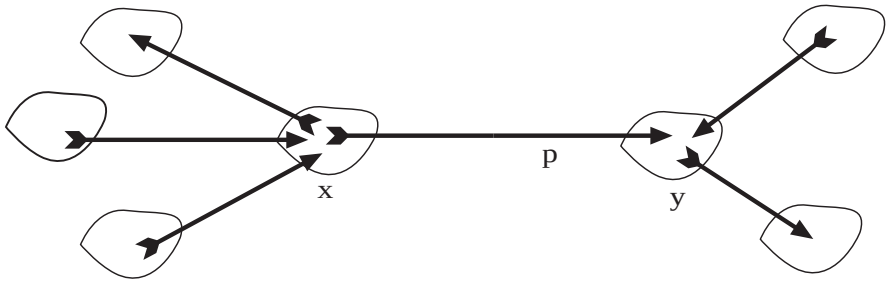
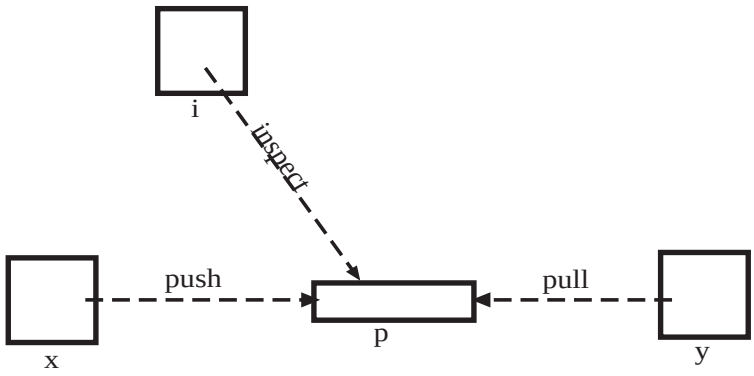


Figure 8. Two Divisions, with Two Different Views of a Base Object



(a) A network



(b) The Sharing of Pipe p

Legend:
pipe:
pointer:

Figure 9. Modeling a Network