An Object-Based Programming Model for Shared Data

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
kaiser@cs.columbia.edu

Brent Hailpern*
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY 10598
bth@ibm.com

21 December 1990

Abstract

The classical object model supports private data within objects and clean interfaces between objects, and by definition does not permit sharing of data among arbitrary objects. This is a problem for real-world applications — such as advanced financial services and integrated network management — where the same data logically belongs to multiple objects and may be distributed over multiple nodes on the network. Rather than give up the advantages of encapsulated objects in modeling real-world entities, we propose a new object model that supports shared data in a distributed environment. The key is separating distribution of computation units from information hiding concerns. Minimal units of data and control, called *facets*, may be shared among multiple *objects* and are grouped into *processes*. Thus a single object, or information hiding unit, may be distributed among multiple processes, or computation units. In other words, different facets of the same object may reside in different address spaces on different machines. We introduce our new object model, describe a motivating example from the financial domain, and then explain facets, objects and processes, followed by timing and synchronization concerns.

*Hailpern's current address is IBM, Old Orchard Road, Armonk, NY 10504.

keywords: database, distributed system, financial application, programming language design, rapidly changing data, sharing

1. Introduction

The classical object model [Wegner 87] supports private data within objects and clean interfaces between objects. The standard mode of communication between objects is message passing, where a client object sends a message to a server object to request some services defined in the server's interface; the client is not aware of the private data hidden within the server and cannot manipulate this data except through side-effects of the server's responses to its messages. This encapsulation makes it impossible for two or more arbitrary objects to transparently and symmetrically *share* data in a tightly-coupled manner. This is reflected in both the compile-time view of objects as information hiding units and their execution-time view as processes. Our goal is to extend the classical object model to support transparent, symmetric, tightly-coupled sharing.

The classical object model permits code and data *definitions* to be shared via inheritance, but not data *values*. Data values can be shared symmetrically but non-transparently, by encapsulating the shared data in a third object accessed through message passing [Agha 86]. Data can be shared asymmetrically by encapsulating it within one of the objects, which may access it transparently, but the only access available to other objects is through message passing; any sharing is by convention and outside the programming model. It is possible to share data symmetrically and transparently, but only among all instances of a certain class (class variables) or a certain set of classes (pool variables), or all objects (global variables) [Goldberg 83]. We know of one previously proposed object model that does support shared data among arbitrary objects: Self [Ungar 87] treats all data as potentially shared, transparently and symmetrically, but the data is loosely-coupled. There is no language support for compile-time consistency checking of access to any data; instead, all access is through message passing.[1]

We are concerned with distributed applications, where clients and servers may reside in the same process or in different processes on the same or different machines. Distribution complicates sharing because two objects that share a common subpart need not reside in the same address space, and in these cases apparently direct access must be implemented via message passing. A few distributed object systems (e.g., [Wedde 89, Shapiro 89]) support some form of partitioning of object subcomponents among different nodes, but the subcomponents cannot be shared among multiple objects.

---

[1]A recent paper on Self [Chambers 89] describes an optimizing compiler that, among other things, infers types and uses this information in code generation.

We propose a new object model that supports data sharing among arbitrary objects in a distributed environment by separating compile-time and execution-time concerns. There are three important components: *facets*, *objects* and *processes*. Facets are the minimal units of data and control; facets may be shared among multiple objects and reside in processes. Our objects reflect the compile-time aspect of classical objects: each object encapsulates one or more facets and provides an external interface to other objects. Our processes reflect the execution-time aspect: each process co-locates one or more facets within a single address space and manages their scheduling (i.e., allocation of computation and communication resources). Objects and processes are orthogonal: objects are not contained in processes nor vice versa.

Our new model is motivated by an important application domain, financial services. Advanced financial services, e.g., ‘‘programmed trading’’, involve: (1) enormous amounts of data; (2) sharing of data among large numbers of simultaneous users; (3) rapidly changing data, for example, as prices of financial instruments fluctuate; (4) changes to data outside the control of the system, for example, from the stock exchange wire; and (5) economic penalties for making decisions based on obsolete data. These problems have been articulated by other researchers, but not solved (e.g., [Peinl 88]).

Objects are naturally suited to represent such real-world phenomena, except that there is no provision for sharing data among objects. This is the motivation for our extension of the classical object model to support transparent, symmetric, tightly-coupled sharing of data. Transparency is needed since the shared data is logically part of each sharing object. Symmetry is needed to treat multiple sharing objects uniformly. Tight coupling is required to guarantee static semantic consistency at compile-time. Tight coupling is also important at run-time, since our rapidly changing data is similar to the real-time data of manufacturing and telecommunications: the data changes when it changes, and cannot be blocked until convenient. However, financial applications are more like telecommunications than manufacturing, since performance can degrade gracefully (to a point) as changes arise on the order of seconds rather than microseconds and sampling is acceptable.

Financial services is also one of the primary motivating applications for database management systems, which support shared data (the database) and separate compile-time (data definition) from execution-time (database applications) issues. But conventional databases do not support encapsulation within objects, or even a clear notion of ‘‘object’’, while object-oriented databases

[Dittrich 86] do not support sharing among objects. Neither makes execution-time issues such as scheduling and data placement among nodes explicit in the programming model, but buries concurrency and distribution in the underlying database manager. Our object model addresses these problems at an abstract level, although it does not treat the persistence or associative query issues of databases. Our model is not specific to financial services, but is suitable for other applications with similar requirements, such as network management [Mazumdar 89], weather modeling and animation [Haeberli 88].

We start by discussing an extended financial services example in general terms to motivate our new object model. Then we give an overview of a new programming language, PROFIT (PROgrammed FInancial Trading), based on our model. PROFIT is intended as a *coordination* language [Ciancarini 90], with local declarations and statements written in a *computation* language, such as C or C++. We then present PROFIT's facets, objects and processes, followed by a discussion of timing and synchronization concerns. Two prototype implementations of subsets of PROFIT have been developed, and both are discussed briefly. We compare our work to related work, and summarize our contributions.

2. Example Portfolio Management System

Consider a financial market, with both stocks and options, collectively called instruments. Our system manages portfolios made up of combinations of such instruments. For the sake of the example, we assume there are only three companies, *Institutional Books and Materials*, *Domestic Educational Corporation* and *Supplies, Umbrellas and Novelties,* abbreviated INS, DOM and SUP. Options on their stocks are available with various expiration dates. This system, called the *Stock Environment Calculator* (SEC), monitors the current prices of the stocks and options and executes the appropriate purchases and sales (according to certain constraints associated with the particular portfolio by a financial analyst) as market conditions change. We describe three aspects of SEC: consistent access to the current prices of a stock and its options (the 9am stock price along with the 9:10am option price could result in disastrous financial strategies); easy programming of the code that tracks the changing prices to determine when action should be taken; and transparent sharing of the prices by all the users of the system.

The key challenge is that the prices of the various stocks and options change rapidly, perhaps several times a minute, and to a first approximation independently from each other and from the actions taken within SEC.[2] What we have in mind is essentially soft real-time processing, where data can be sampled at particular time intervals (typically seconds rather than microseconds) and the precision of the time intervals may be coarse (i.e., we do not assume a real-time operating system). Another difficult problem is to support many simultaneous users of SEC, transparently and efficiently sharing the rapidly changing data under these soft timing constraints.

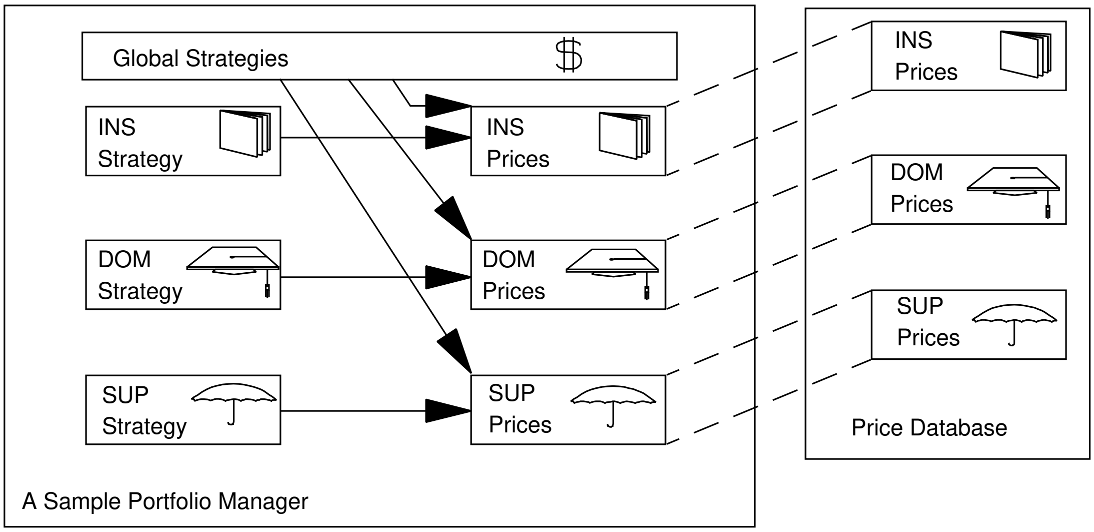

Figure 2-1: Example Portfolio

An individual portfolio consists of a set of specific instruments and a set of strategies for when to buy and sell (for simplicity, we assume strategies are written as simple imperative programs — for example, ‘‘buy low, sell high’’ — that do not change over the life of the portfolio).[3] Each portfolio is organized into subparts, where each part represents the instruments for a particular company, together with any aspect of the strategy specific to that company (see Figure 2-1). For example, if the price of a particular company’s stock increased 10% since purchase, or over the last 24 hours, buy more (or sell). Additional subparts support strategies that cut across companies and maintain any other data needed. A subpart representing the instruments of a particular company contains the current price of the stock and the current price of each option (we assume in our examples only two options per company, expiring at different times with the same strike price). Multiple portfolios will refer to the same company’s instruments with

---

[2]We assume that most buys and sells continue to be executed outside the PROFIT application system.

[3]As part of his MS thesis at Columbia University, Tushar Patel is interacting with Citicorp to determine a realistic approach to defining strategies.

independent criteria for when price changes are significant to the financial analysts' strategies. The difficult problem is how to notify the computations reflecting the strategies of these portfolios when the prices of the stocks and options change. There are three ways to structure the solution. The *active value* approach propagates each change to every interested portfolio. *Polling* requires every interested portfolio to poll the current value. The third, preferred approach uses *daemons*. In this section, we compare these three approaches at an abstract level, and describe PROFIT's support for the daemon approach in the remainder of the paper.

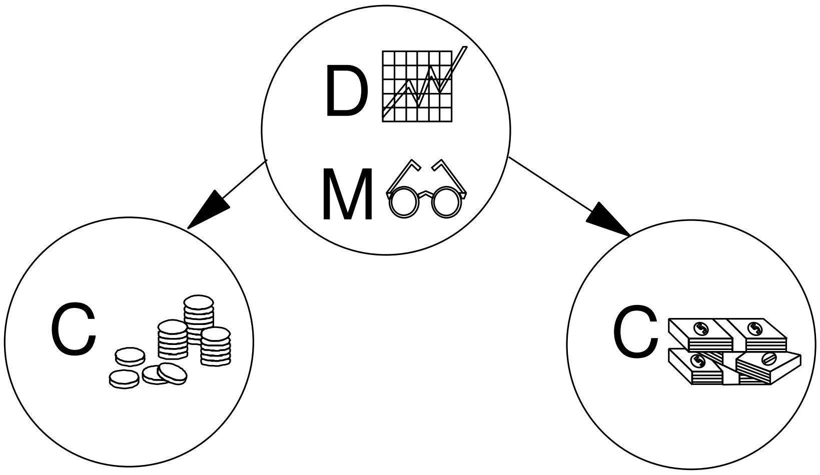

Figure 2-2: Active Value Approach

In the active value approach, supported by several object-based programming languages [Stefik 86], any change to any value is propagated to all other parts of the program that might be ''interested'' in that change. The propagation invokes code associated in advance with the data and the kind of change. A traditional application is changing the speed of a simulated car, with propagation resulting in an updated display of the speedometer reading and the consumption rate of fuel. As shown in Figure 2-2, the active value approach combines the shared data (D) and whatever computation is necessary to monitor (M) the changes to the data in the same entity.[4] The monitoring is accomplished via side-effects of the procedures that store (or retrieve) the data. The monitor code has the responsibility to notify all other interested computations (C) of changes.

The active value approach has a significant flaw with respect to our application: since price

[4]An ''entity'' (one of the circles in Figures 2-2, 2-3 and 2-5) is not necessarily equivalent to a ''process''. This is explained in detail in Section 6.

changes are frequent and typically small (1/4 point), some changes may be insignificant from the point of view of some portfolios. Thus the system can be flooded by many notification messages to which few portfolios are paying attention: the consequent network traffic may be unacceptable for real-world scenarios. One solution is for the monitor code to know the separate criteria for each interested portfolio regarding what changes it considers important. This would add a substantial computational component to the active value and greatly complicate the programming of the monitor code, as well as adding significantly to processing overhead. (This problem is even more significant when new portfolios can be added while SEC is in operation. We discuss dynamic reconfiguration elsewhere [Hailpern 90] and do not address this topic in this paper.)

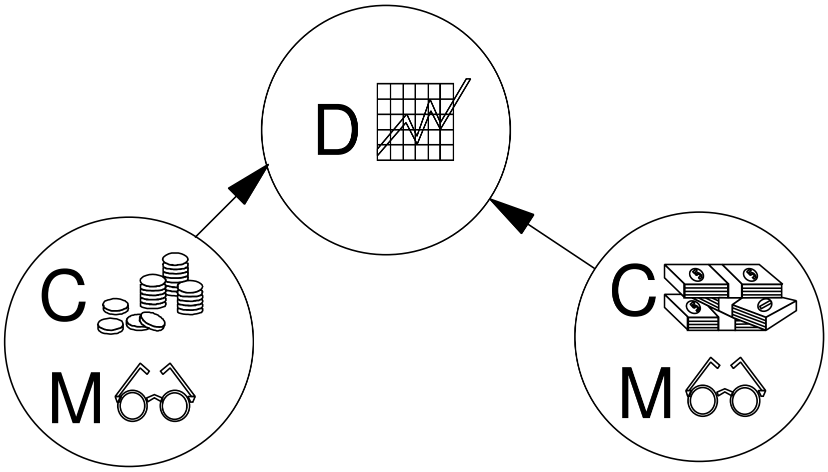

Figure 2-3: Polling Approach

Polling is the traditional means for implementing device drivers, and can be implemented in any imperative programming language. The idea is to check over and over again whether a given data item has changed. Figure 2-3 illustrates this approach: the shared data (D) is passive, and the interested computation entities (C) directly incorporate the monitoring (M) of changes to the data. The computation code must include explicit statements to check whether the shared data has been changed in a manner considered significant by the particular computation.

Polling overcomes the problems with the active value approach, since each portfolio can decide how often to check each stock or option price and under what criteria to take action. Unfortunately, this requires the programmer to manage computation resources (processor time) and communication resources (network bandwidth) by hand, in deciding where the main strategy should be interspersed with which polling code. This is not a serious impediment when polling

```
INS_up_10%:
    {sell_INS();}
INS_down_15%:
    {buy_INS();}
INS_exceeds_DOM:
    {buy_DOM(); sell_INS();}
```

without explicit polling

```
int old_INS, old_DOM;
    while (1) {
        if (INS_price>(1.1*old_INS))
            {sell_INS();old_INS=INS_price;};
        if (INS_price<(0.85*old_INS))
            {buy_INS();old_INS=INS_price;};
        if (INS_price>DOM_price && old_INS<old_DOM)
            {buy_DOM(); sell_INS();
            old_INS=INS_price;
            old_DOM=DOM_price;};
        }
```

with explicit polling

Figure 2-4: Complexity of Polling

only a single price, but is complicated when the latest prices of multiple instruments must be considered both individually and in combination [Faulk 88]. Figure 2-4 compares a very simple case involving only two instruments with all polling at the same frequency.

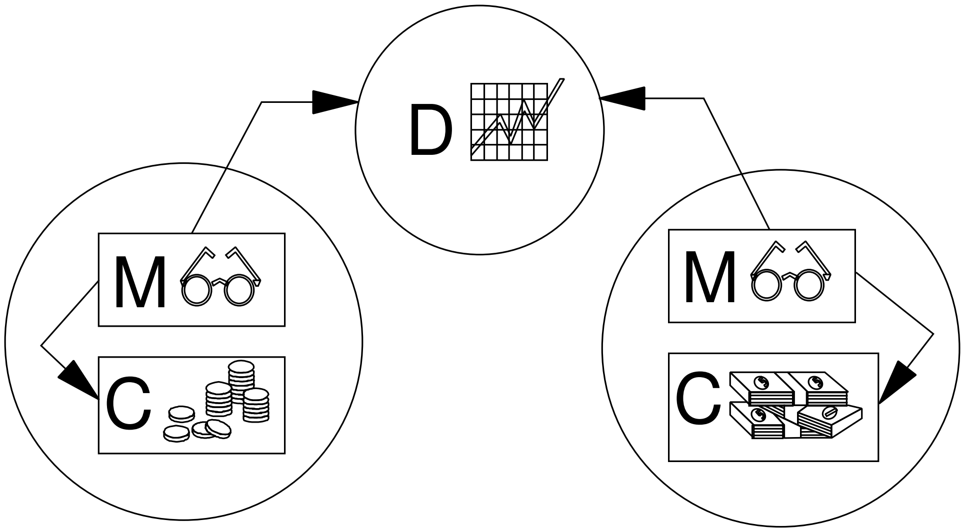

Figure 2-5: Daemon Approach

We prefer a more sophisticated solution in which the shared prices are monitored by simple daemons [Bach 86]. Each daemon contains a trigger that informs the main-line strategy when something ‘‘interesting’’ happens. Active values and polling both employ two entities, one representing the prices and the other the portfolio manager. The daemon approach uses three entities, adding an intermediate entity that polls the prices and behaves as an active value with respect to the portfolio manager. The advantages include a simpler programming model and the capability for each user to decide how best to balance computation and communication resources between the three entities.

For example, one daemon might poll more frequently than another if the first financial analyst is concerned with up-to-the-minute prices while the second bases decisions only on long-term trends.[5] Figure 2-5 shows the three entities: passive shared data (D) as in polling, the strategic computation (C) as in the active value solution, and a separate daemon (M) that both polls the data and informs the associated portfolio manager of important changes. This makes it easy to program special monitor code for each computation without disrupting the programming of the computation code (i.e., the code implementing the financial strategy). Further, it may be possible to lower communication costs if the daemons can be placed within the prices process(es), or perhaps more practically, in separate processes on the same machine as the prices (i.e., the machine connected to the stock prices feed), while the strategies might execute on users' workstations.

Daemons reflect an intermediate point in the spectrum between polling and active values, and capture the best of both. A daemon can use more complex criteria than either polling or active values. For example, it can decide how often to poll based on volatility of the market, and it can propagate changes based on the specific criteria provided by its client portfolio manager, which might change over time.

One way to aid the financial programmer in applying this system architecture based on daemons would be to implement a library of subroutine calls in some conventional programming language, such as C or C++. However, it seems unlikely that any one daemon solution as coded would work across all programming problems and/or machine architectures. Some programming problems might be better served by polling or active values, while others might be able to exploit a different combination of polling and active values than what we have defined in our daemons.

Therefore, we have designed a programming model to provide the appropriate building blocks to easily represent the daemon solution as well as active values and polling. Now the question arises: What is the best way to present the components of this model? The building blocks could be reflected by collections of subroutines called from an existing language or integrated into a

[5]The first analyst is presumably willing to pay for the additional resources consumed; real economic costs are considered in Section 7.

uniform level of abstraction as part of a new programming language. We chose an intermediate point on this spectrum: we maintain traditional data structures and control flow from a conventional computation language, but add pervasive new constructs representing our programming model to coordinate among computations (the term ‘‘computation’’ as used here should not be confused with the computation entities described above, and in fact encompasses computation, monitor, data and other entities). Relying on an existing computation language allows us to focus on the innovative aspects of our model. An additional advantage of this language design technique is that prototype implementations can be written as simple preprocessors plus run-time libraries rather than full-scale compilers (although this introduces the canonical problem of debugging a preprocessed language)[6].

Our contributions are an explicit programming model for *shared data* in a concurrent object system and an associated programming language. The model provides transparent, symmetric and tightly-coupled sharing by decoupling information hiding and computation/communication concerns. In the sequel, we explain how the components of our programming model, as expressed in the programming language, directly address the implementation of our example and support other applications with the characteristics outlined in the Introduction.

3. Overview of PROFIT

Our programming model reflecting the implementation architecture sketched in the previous section motivates our design of the PROFIT programming language. There are three main concepts:

- *Facet*, the minimal unit of data and control, in particular, the unit of shared data. A facet consists of a number of named slots, each of which may contain either a data value or procedure code. Each facet includes a special `active` slot, which indicates the currently running operation (evaluation of a data or procedure slot); operations may not execute concurrently within a facet.[7]
- *Object*, a statically defined collection of facets representing an information hiding unit. An object defines a context for binding references between facets in the same object and an external interface for receiving messages from other objects.
- *Process*, a statically defined collection of facets — orthogonal to objects — that

[6]Debugging methodologies and tools for applications involving rapidly changing data is itself an interesting research topic, but outside the scope of this paper.

[7]A facet is superficially similar to the ‘‘frame’’ of AI languages [Charniak 80], but analogies based on frames would be misleading.

must execute at the same physical location. That is, a process represents a single virtual address space in which its facets reside, and allocation of computation and communication resources to facets is handled by their process. Multiple facets may execute concurrently within the same process.

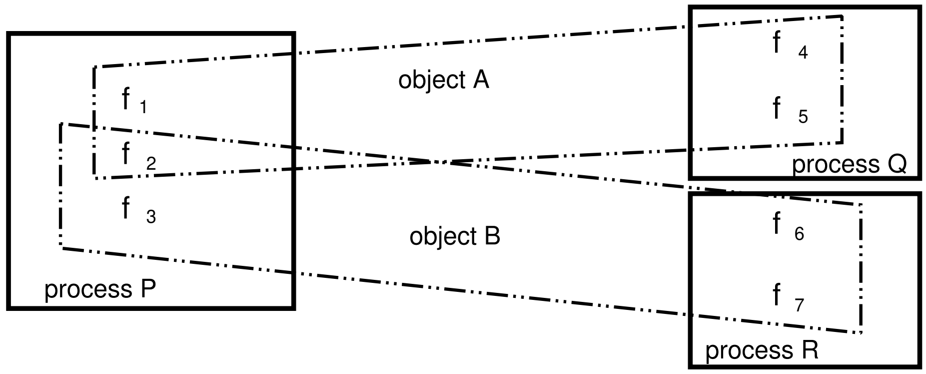

Figure 3-1: Facets, Objects and Processes

Every facet is a member of one or more objects and exactly one process. In the simplest case, there is exactly one process in the entire program, and thus all objects execute on the same machine within the same virtual address space. When there are multiple processes, different facets of the same object may reside in different processes. Objects and facets can be written independently of processes, and later configured into a system by defining processes; any legal configuration will provide a logically correct program, although one may be more efficient than another on a given concurrent architecture [Coffin 89].

What we have in mind for the example financial application is one SEC process containing all the prices, and one additional process for each individual user of the SEC.[8] Each user would define one object corresponding to each of his portfolios, each object consisting of new computation and daemon facets and the relevant previously defined prices facets. The computation facets would live in the user's process, while the prices facets would be updated only in the SEC process; the daemons could live in the same process with the prices, allowing tighter timing constraints and lower communication costs than if they lived with the portfolio computations, but there are difficulties incurred by security and other proprietary concerns, not addressed in this paper. This relationships between facets, objects and processes is illustrated in Figure 3-1. These concepts are explained in more detail in the next three sections, while timing

[8]Actually, a user might own multiple processes, each for a different portfolio.

and synchronization issues are considered in the following section.

There is a fourth concept, *Program*, that specifies the objects and processes that together make up a single executable program, the physical locations of the processes at execution-time and the initialization code to start the program running. For the purposes of this paper, we assume that facets, objects, processes and programs are all defined statically, so it is not possible to add new components while a program is executing. In the more general case, however, it would be necessary to be able to add user-defined objects (portfolios) and corresponding processes (executing on behalf of new users) to an already-executing program, and be able to remove such objects and processes (e.g., when their subscription to the SEC service ends), as well as adding and removing instruments.

PROFIT's facets and objects provide nice abstractions for programming each of the three approaches to our example of the previous section:

- In the active values approach (Figure 2-2 in the previous section), a prices facet is shared among several objects. This facet must provide the data, change monitoring and notification of interested objects. The facet must be balanced (by its programmer) between computation, to decide which objects are interested, and communication, to make sure all interested objects are informed of changes. Each of these objects also contains one or more non-shared facets that receive the notification and carry out the appropriate financial strategy.
- In the polling approach (Figure 2-3), the prices facet contains shared data only. Each interested object includes a non-shared facet that carries out both computation and change monitoring for each prices facet, with an ad hoc balance between how frequently change should be polled for and the disruption of the primary computation.
- In the daemon-based approach (Figure 2-5), the prices facet again contains only shared data. Each interested object includes (at least) two non-shared facets, one that carries out the computation and the other that does the change monitoring. Each change monitoring facet is devoted to polling a shared prices facet, performing only that computation necessary to determine which changes are interesting to its object, and notifying the strategy facet accordingly.

Processes support the run-time behavior of our example: All relevant timing constraints are expressed directly by the procedures located within a facet's slots, so each facet can poll/notify at the time intervals appropriate for the object(s) containing it. It is the responsibility of the enclosing process to schedule execution of facets to meet these timing constraints. Timing issues are elaborated in Section 7, and until then we are concerned primarily with the expressiveness of PROFIT's programming model.

As mentioned in the introduction, PROFIT is more a language extension than an entirely new language, in that it does not define the details of the base language, in this case the data and procedures that may appear in slots. We intend these to be written in a conventional programming language, perhaps multiple languages, but for discussion purposes we assume C.[9] Thus data slots contain C data values and procedure slots contain C functions. A PROFIT program would be translated automatically into C augmented with a package of macros and subroutines. This package would provide facilities for evaluating slots within the same facet, handling indirection to other facets in the same object and to the entries in the external interfaces of other objects, and so on.

One issue is whether multiple data slots, of the same or different facets, can point to the same data structure. This is a problem since different facets might reside in different processes, so direct sharing is not always possible. One solution might be distributed virtual memory [Li 86]. We follow a simpler approach: no sharing of data structures, only entire facets (i.e., there is no pointer aliasing). Data structures are copied when transmitted as arguments (often called marshaling [Gibbons 87]), whether within the same process or across process boundaries.

4. Facets

The facet is the minimal unit of data and control. A facet has a unique name and a set of named slots, each of which may contain either a data value or procedure code. Slots are typed, either the type of the data (a C datatype) or the return value of the procedure (a C datatype or `void`). Procedure slots must be equated to specific C functions at compile-time, while data slots may optionally be assigned to a specific value determinable at compile-time or run-time; if not explicitly initialized, data slots are implicitly set to null values. Evaluating a data slot returns the current value, while evaluating a procedure slot executes the procedure (with the parameters provided) and returns the result of the execution, if any. Data slots may be reassigned during execution to new values, but procedure slots cannot be changed.

For example, the set of prices for the instruments of the Institutional Books and Materials company would be represented as a facet, called INS-instruments (Figure 4-1). The only operations are (implicit) `get` and `put` pairs for each data slot. In this example, there is one

[9] We have developed two preliminary implementations, one for C and the other for C++, discussed in Section 8.

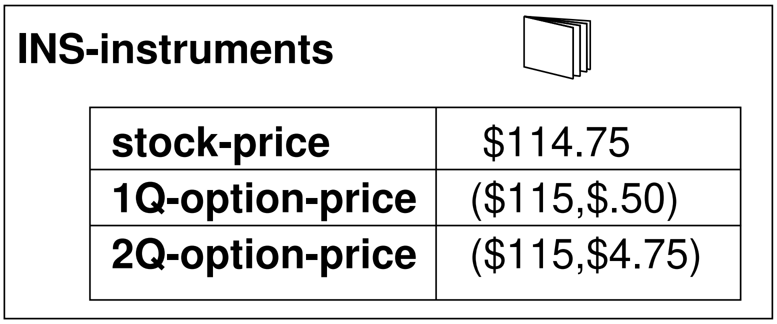

Figure 4-1: INS-instruments Facet

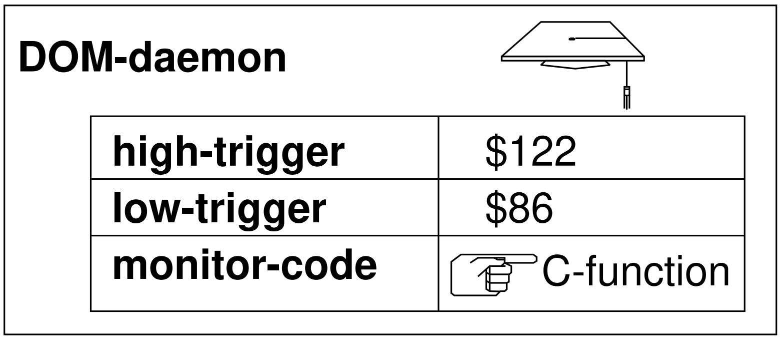

Figure 4-2: DOM-daemon Monitor

possible writer — some agent external to SEC representing the stock prices feed — and multiple readers (daemons) from different portfolios. In the general case, however, facets permit read/write sharing. Another example facet would be the daemon that monitors the changing prices of Domestic Educational Corporation's instruments (Figure 4-2). The daemon would keep certain local data such as high and low trigger values, used by its procedures (here `monitor-code`) when deciding whether to notify the corresponding portfolio manager.

There is a distinguished slot within each facet, called `active`, that represents the currently executing operation (either evaluation of a procedure slot, or `put` or `get` on a data slot). By definition, there can be at most one operation executing within a facet at any time, hence all reads and writes are mutually exclusive. Among other things, the operation indicates its originating object (the notion of originating object is explained in the next section). If the `active` slot is null, then the facet is not doing anything.[10]

---

[10]The `active` slot is not shown in the figures in this section, to concentrate on data and procedure slots, but is illustrated in some figures in the next section.

A prices facet, for example INS-instruments, would normally be passive except while `get` or `put` is running. The `get` operation must be able to return multiple prices from the INS-instruments facet to ensure consistent access, while for our example the `put` operation could be restricted to a single value since price changes are independent (i.e., price changes are real-world, and provided from outside the SEC system)[11]. A portfolio management strategy, say involving statistics specific to the Institutional Books and Materials company, might be passive until the daemon wakes it up after a significant change to decide whether to buy or sell, or might be continuously running some procedure that carries out some extensive analysis.

A facet represents a closed scope, meaning every use of an identifier matches an identifier defined within the facet. There are no free variables. Each procedure is also a scope, with its own local variables, although it is not closed. The facet's slots are global to the procedure, and the facet is treated as the procedure's enclosing scope. A procedure can access only its local variables and the slots of its facet, not anything outside the enclosing facet.

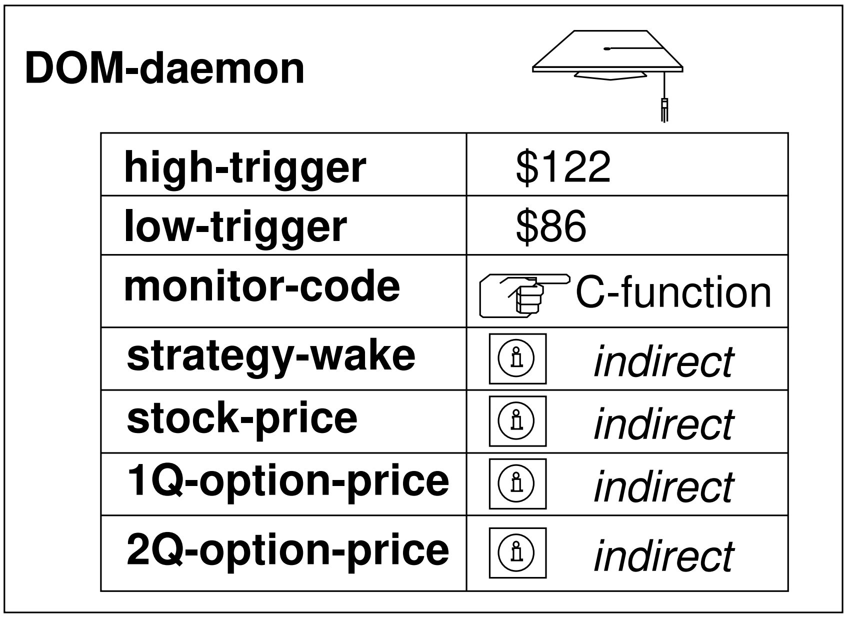

Figure 4-3: Indirect Slots

In order to support references from one facet to another, one or more slots of the first facet may

[11]The instrument facets represent the real prices. Analysis of hypothetical market conditions would involve ''what-if'' prices represented separately.

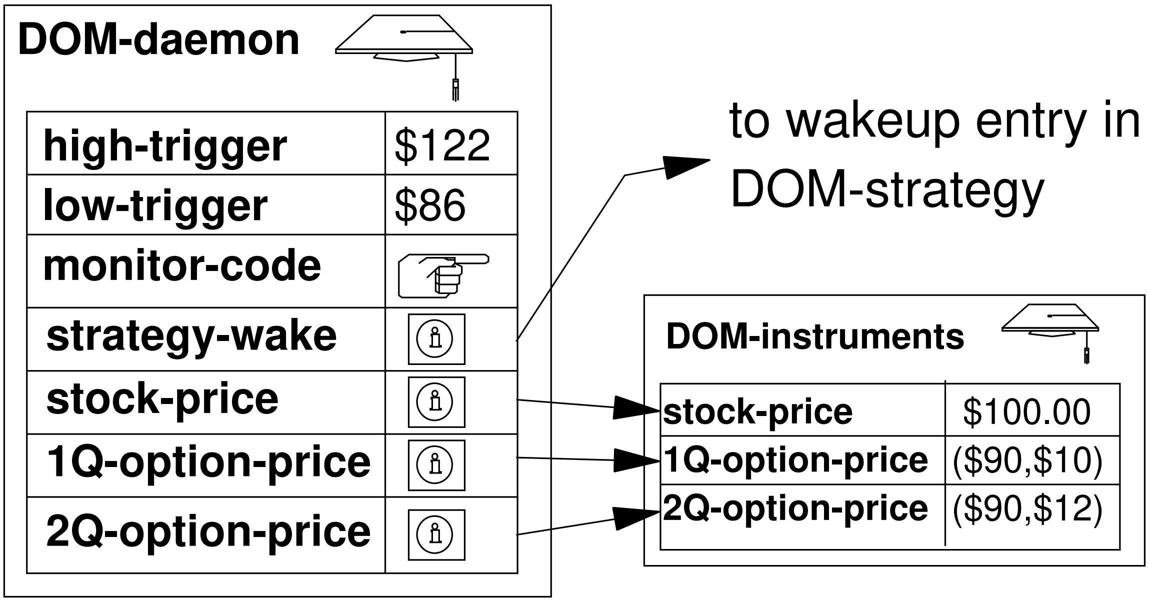

Figure 4-4: Binding

be declared ***indirect***, as depicted in Figure 4-3.[12] The containing object is then obliged to provide a *binding*, normally to a slot in some other facet within the same object (the other possibility is to an entry in the interface of another object, discussed in the next section). Every object has a *binding table* for this purpose. When a procedure slot is being executed, and the code references an indirect slot (data or procedure) of its facet, then the semantics are to refer to the current object's binding table to resolve the reference. (There may be more than one containing object, so the notion of *current* object is significant; this is explained in the next section.)

This approach is related to the *compositional model* of the Jade programming environment [Raj 89], where each component (which may be an object, operation, type, etc.) is associated with a ‘‘habitat’’ that specifies all external dependencies. Objects are constructed from components by matching the facilities provided by one component with the dependencies of another. Our approach is closer to *delegation* than to either this compositional model or the traditional inheritance of object-oriented languages. In delegation, when one object cannot handle a message, it defers it to another [Lieberman 86, Stein 89]. We diverge from the traditional notion

[12]In this figure, the indirect slots are labeled with both the indirect symbol, an "i" in a circle, and the keyword ***indirect***, but later on only the symbol is used; similarly, procedure slots representing C functions are labeled with the function symbol, a pointing finger.

of delegation by resolving indirect references on a slot by slot basis rather than an entity by entity basis; the actual resolution takes place at the enclosing level (object) rather than at the entity level (facet). Further, we delegate fully to the receiving facet's own context (its own slots) and it cannot access slots in the delegating facet (except through its own indirect slots), i.e., there is no ‘‘self pointer’’. This is relevant if the evaluated slot is a procedure that references other slots. Our rationale for this decoupling is that facets should be self-contained, and written independently of any other facets; they may assume external code and data, to be provided by the enclosing object, but must explicitly declare these assumptions. Figure 4-4 illustrates the resulting binding for the DOM-daemon. Details are discussed in the next section.

5. Objects

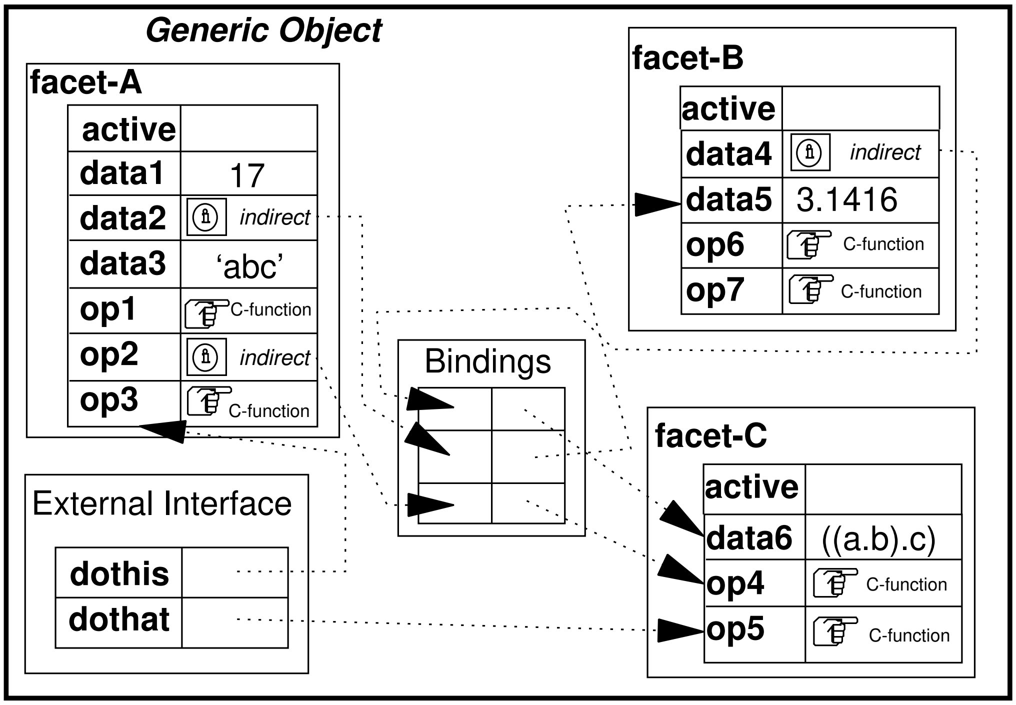

Figure 5-1: Generic Object

As in the classical object model, a PROFIT object is the compile-time unit of information hiding. It defines an external interface and encapsulates its internal data and procedures. The interface defines the object's unique name and the set of entries visible to other objects. PROFIT objects are different from classical objects in that all internal data and procedures are supplied by facets. An object encapsulates its member facets in that all external references must go through the

Binding Table

(facet-A,data2)	(facet-B,data5)
(facet-A,op2)	(facet-C,op4)
(facet-B,data4)	(facet-C,data6)

Figure 5-2: Generic Binding Table

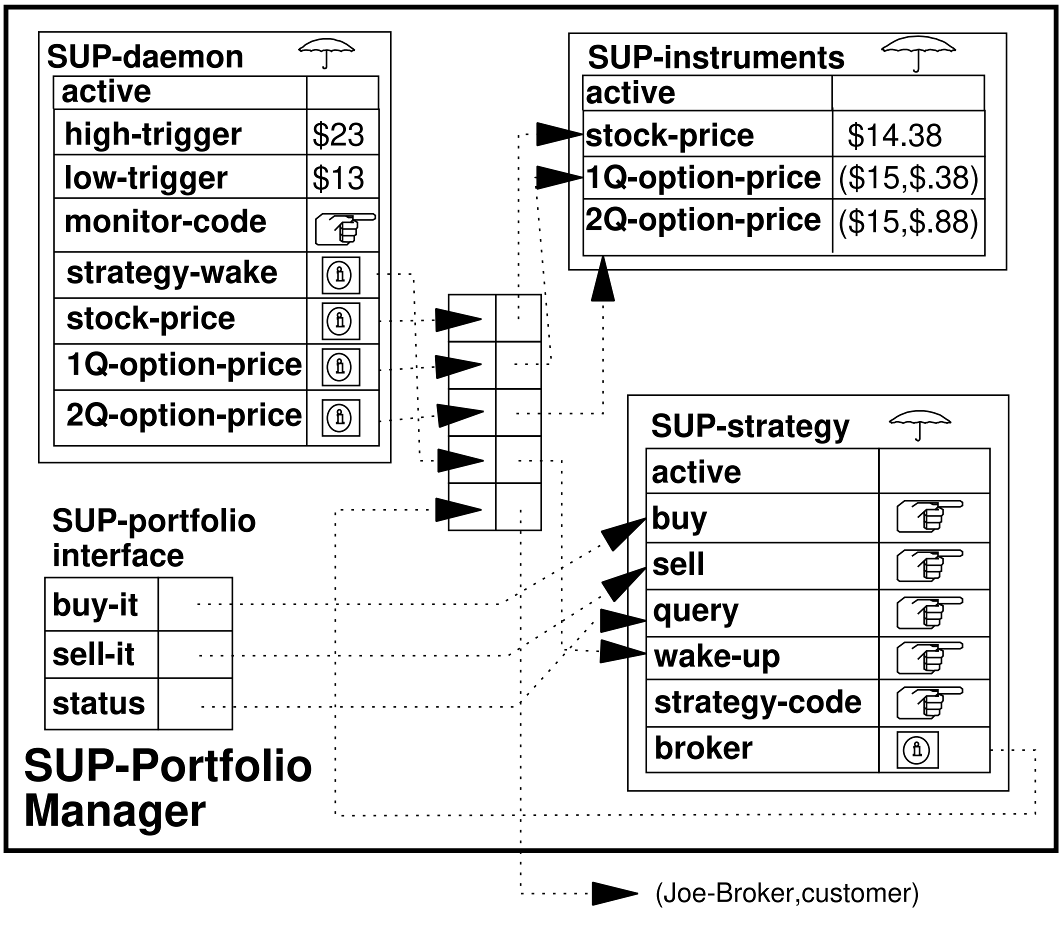

Figure 5-3: SUP Portfolio Object

object interface. This means that each object must bind each entry in its interface to a slot in one of its facets. Since facets may contain indirect slots, each object must also bind each indirect slot in one of its facets, either to a slot in another one of its facets or to an entry in the interface of another object. In both cases, the result of the mapping is to a pair, either (facet, slot) or (object, entry). A generic object is shown in Figure 5-1. Figure 5-2 shows how several facets may be bound together within a generic object, and bindings for the SUP portfolio object are illustrated in Figure 5-3. Note in particular the indirection to the `customer` entry in the external interface of the separate `Joe-Broker` object.

When code (a C function) is executing within a facet, it may directly access only those slots defined in the same facet. Accesses to indirect slots are resolved through the binding table at run-time.[13] Evaluating a slot results in a call to a C function, for a procedure slot, or to the `get` or `put` operation, for a data slot. A call within a facet is treated like a conventional procedure call, except that the called operation is (conceptually) placed in the `active` slot.

For calls between facets, we consider first the viewpoint of the called facet, and then discuss the calling facet. There is a queue associated with each facet, and an arriving call is inserted into the end of the queue.[14] When the facet's `active` slot is empty, the facet accepts the call at the front of its queue and sets its `active` slot accordingly. Subsequent indirection is with respect to the binding table of the object responsible for the call (such responsibility is explained later). When the called operation completes, the facet places the response in the queue for the calling facet and clears its `active` slot, and then goes on to accept its next queued call, if any. From the viewpoint of the calling facet, it queues the appropriate operation at the called facet and clears its own `active` slot. The calling facet is not suspended, but may now accept the next call in its queue.[15] When the response to the original call is eventually accepted from its queue, the facet continues with the operation that made the call from the point where it left off.

These discussions of both caller and callee viewpoints are equally valid for indirection to another facet in the same object or to an entry in the interface of another object (and ultimately a facet in this other object). When a call is made across processes, we use the conventional remote procedure call [Birrell 84]. In the next section we discuss the details of implementing these synchronous calls, and also describe mechanisms that permit asynchronous calls.

---

[13]Compile-time optimizations are possible, although complicated by the multiple containing objects, discussed later on.

[14]As we shall see in Section 7, this queue is a priority queue, but for the purposes of the intervening sections we treat it as if it were purely FIFO.

[15]It is important that the calling procedure ensure that its local variables as well as the facet's data slots are in a consistent state before making the call. This issue is discussed further in Section 7 and in the Conclusions.

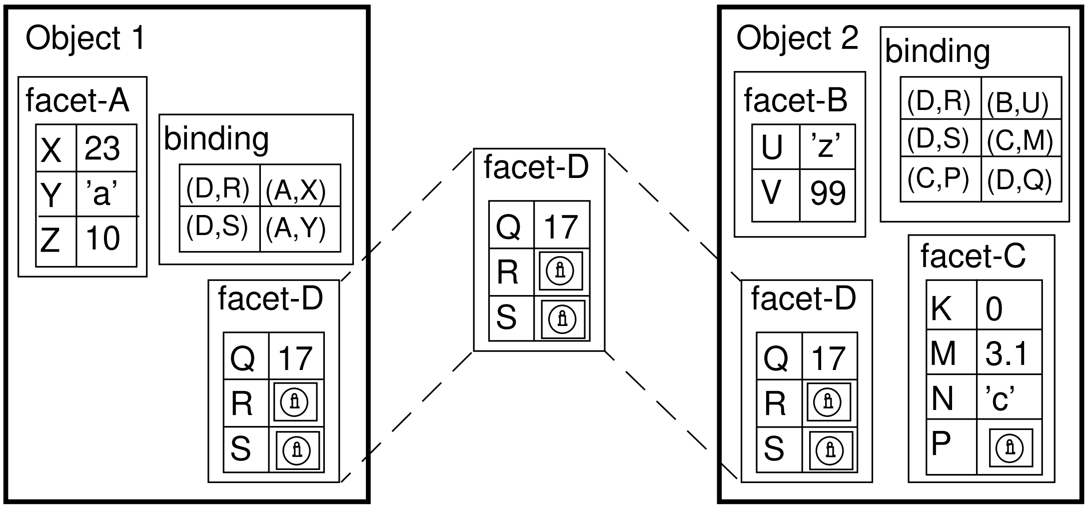

Figure 5-4: Shared Facet

So far, we have considered only the case where a facet is part of exactly one object, and thus there is exactly one binding table that needs to be considered. When a facet is shared among multiple objects, each of these objects provides a *different* binding table that must resolve all the shared facet's indirect slots, as illustrated in Figure 5-4. For example, facet `D`, slot `R` maps to facet `A`, slot `X` in `Object1` but to facet `B`, slot `U` in `Object2`. When a facet is active, only one binding is actually used, the one belonging to the object from whose facet the active facet was called. Since a shared facet may be called from another shared facet, it is necessary for the appropriate binding table (i.e., its pointer or index) to be passed as an implicit parameter in each call.

Since prices of stocks and options are shared among many portfolios, their corresponding price facets are shared among the objects representing the relevant portfolios. Consider two portfolios, Ferdinand and Solid Assets, which both include a number of shares in Instructional Books and Materials. Each portfolio has its own strategy for the buying and selling of INS stocks, but both share the INS-instruments facet, as shown in Figure 5-5.

Communication between objects is a simple extension of the communication between facets. When a call is received at the interface of an object, the object maps the call to a procedure slot of one of its member facets. The call is queued normally at the facet. When the call returns, the object must send the result back to the calling object. Because objects can communicate with many other objects, we associate a queue with each object's external interface. As soon as a call

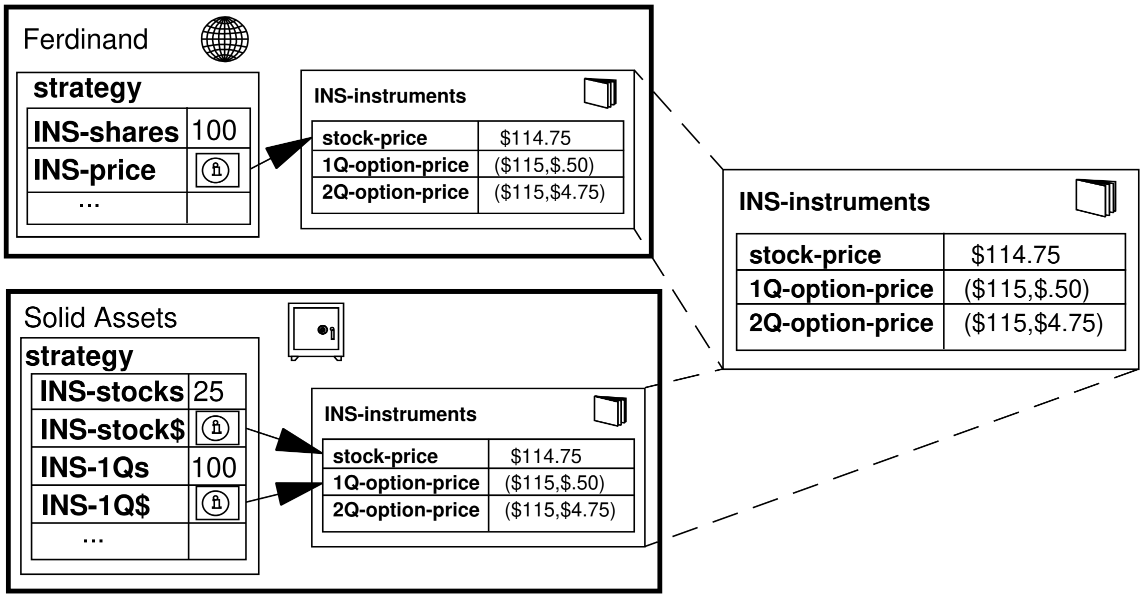

Figure 5-5: Shared INS-instruments Facet

in this queue has been mapped to a particular facet, it is moved from the object's queue to the facet's queue, and the object goes on to resolve the next external call.

6. Processes

The classical object model does not specify a particular role for processes. There is a spectrum of relationships between objects and processes reflected in the concurrent object-based systems described in the literature. An object may be a passive entity manipulated by one or more processes [Dasgupta 88], it may be one of many passive objects contained in a particular process [Strom 84], an object may be an active entity and be equivalent to a process [Detlefs 88], or several objects may reside in the same process and communicate among themselves and/or with objects in other processes [Black 86].

PROFIT processes are close to the conventional notion of processes in operating systems. Each facet resides in the address space of a particular process, and processes thus represent the execution-time organization of facets. In contrast, objects represent the compile-time organization of facets. Objects do not ‘‘live’’ anywhere, and facets of the same object may reside in different processes. The external interface and binding table of every object are replicated in every (relevant) process, with no run-time overhead since unlike the contents of facet data slots, this information is determined at compile-time and is not modifiable during execution.

The significance of a process is in the allocation of its computation and communication resources among the facets that reside in the process. The total processor time of all the facets cannot be more than the processor time allotted by the operating system to the process, and tradeoffs must be made among the computation needs of the facets. Similarly, the total communication of all the facets cannot require higher bandwidth than allotted to the process, and again tradeoffs must be made. (The allotments from the operating system may be fixed or dynamic, but balancing among processes — as opposed to within processes — is outside the scope of this paper.) But before discussing timing and synchronization concerns (in the next section), we first elaborate the implementation mechanisms for synchronous and asynchronous calls among facets, whether in the same or different processes.

The implementation of inter-facet communication is based on *threads*. There are many notions of thread, running a spectrum from light-weight as in Mach's raw threads [Rashid 87] to medium-weight as in Mach's C-thread package [Cooper 88] to heavy-weight as in a full Unix process [Bach 86]. The common concept is that a thread has a single locus of control. What distinguishes the variety of threads is the degrees of ownership of context (registers, stack), data resources (address space), computation resources (processor time) and communication resources (network bandwidth).

What we have in mind is medium-weight threads. Along with a simple locus of control, a PROFIT thread maintains context (that is, a stack) between nested calls, thereby permitting recursion. Alternatively, light-weight threads without their own stacks would necessitate a Fortran-style programming model, without recursion or automatic variables; heavy-weight threads, i.e., one facet per process, would place resource allocation at the operating system level rather than within the programming model.

In the previous section, we discussed the notion of queuing calls for facets and for objects. Here we elaborate in terms of threads. The rest of this section discusses the manipulation of threads in detail. It is important for the reader to appreciate the complexities that are completely hidden from the programmer by PROFIT.

Each call in a facet queue is represented by a thread, which provides the context of the call. Since objects have no physical representation, object queues are implemented as special queues in processes, and processes take over the objects' run-time role in forwarding external calls to the appropriate facets. First we discuss the manipulation of threads in facet queues, and then the creation of threads in response to calls (messages) queued for processes.

When a procedure in one facet makes a call to another facet, the first facet's current thread is suspended and enqueued for the called facet. When a thread reaches the front of a facet's queue, the facet sets its `active` slot to reference the dequeued thread and resumes the thread to evaluate the called slot; when the call completes, the thread is suspended and enqueued for the caller. This works only among facets within the same process, where enqueuing and dequeuing of threads is managed by simple index or pointer manipulation. When calls are made across process boundaries, a stand-in thread must be created in the remote process, as discussed below.

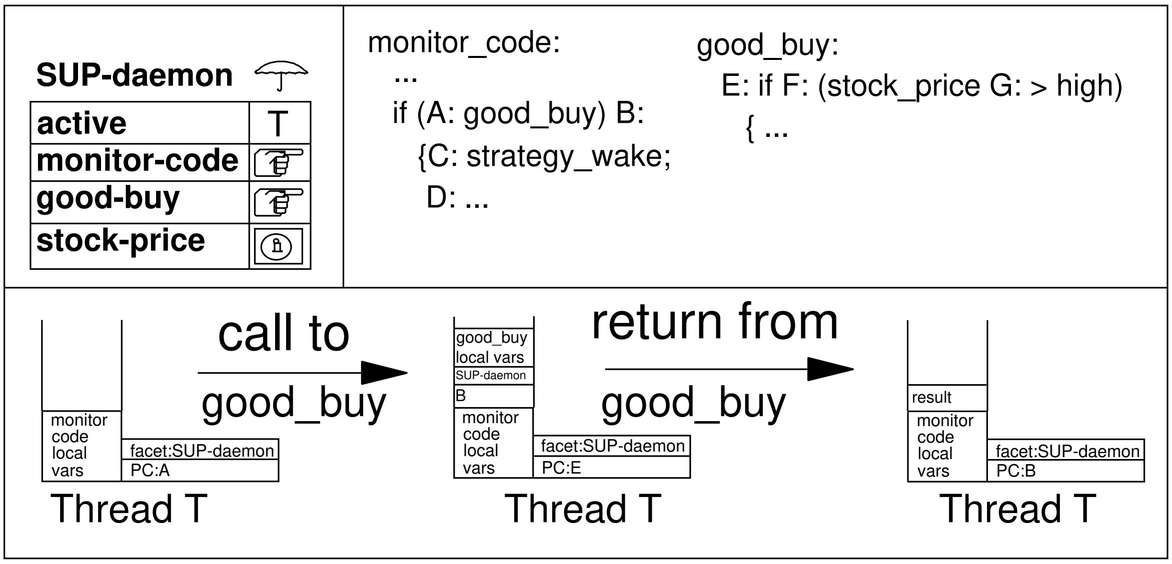

Figure 6-1: SUP-daemon Call Within Facet

Consider the following example of inter-facet calls: The SUP-daemon periodically polls the SUP-instruments prices, to compare to its own trigger values. If the criteria are met, the daemon notifies the SUP-strategy computation. The SUP-daemon's call to evaluate a data slot within its own facet is shown in Figure 6-1, while the SUP-daemon call to get data from the SUP-instruments facet is depicted in Figure 6-2.

There are two situations in which processes create new threads: when a call is made across

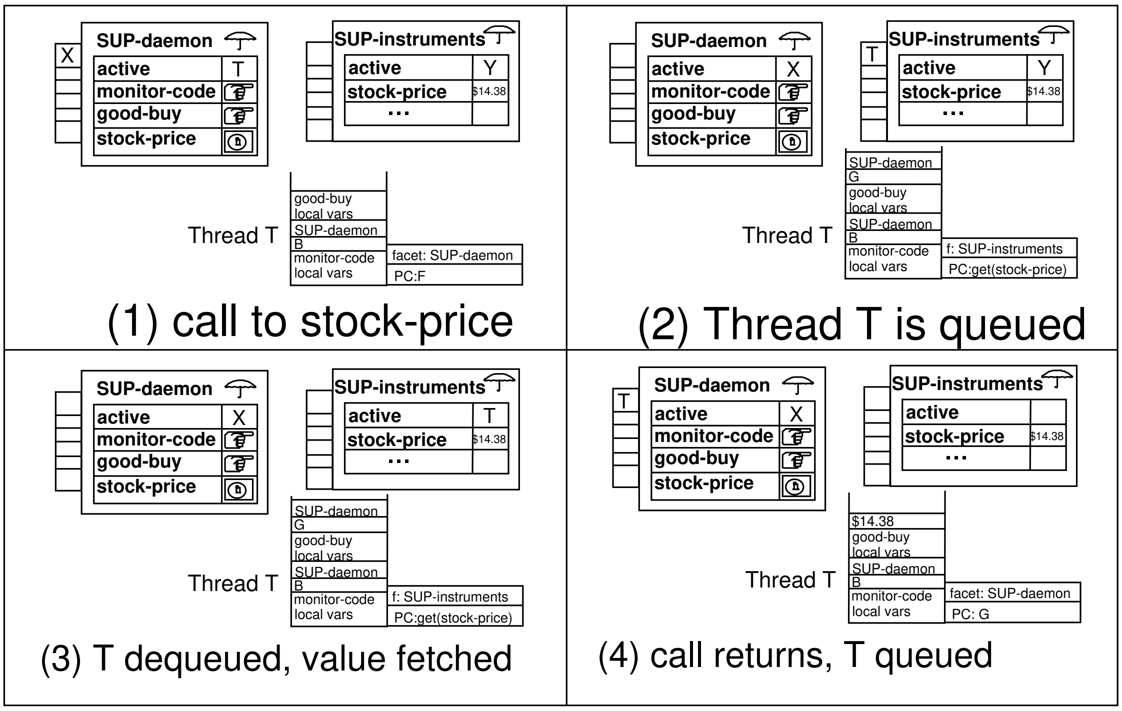

Figure 6-2: SUP-daemon Call to SUP-instruments

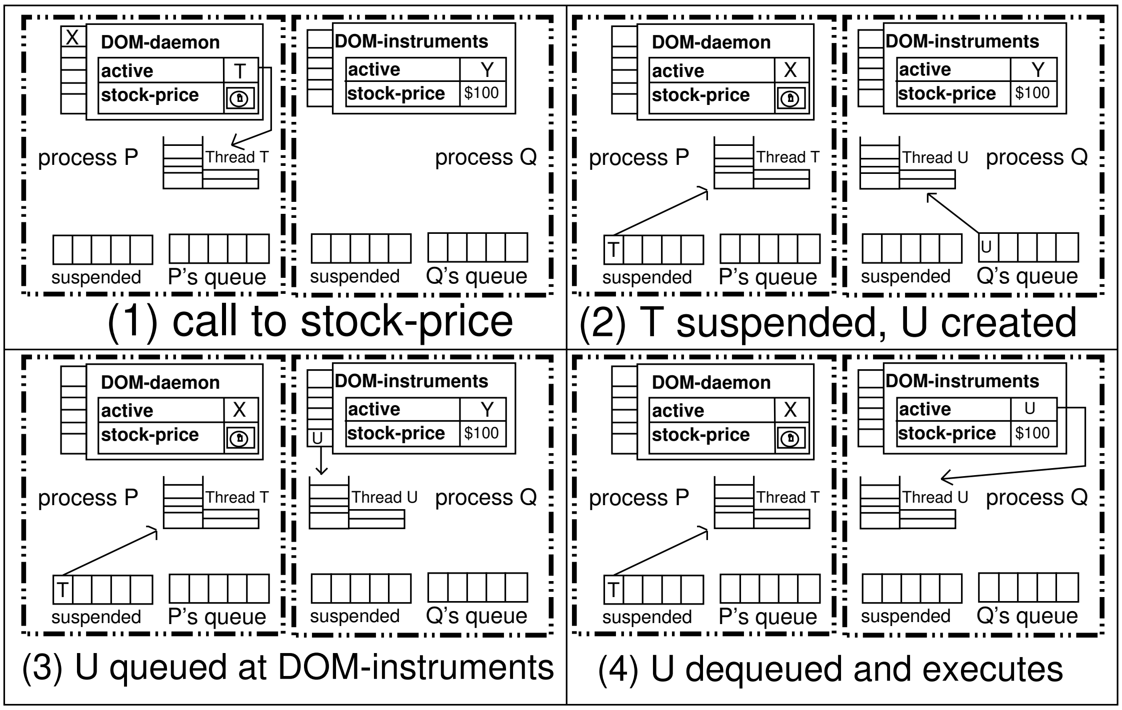

Figure 6-3: DOM-daemon Makes Interprocess Call to DOM-instruments

process boundaries and when an asynchronous call is made, whether within a process or from one process to another. (There is a third case, at system start-up, but we ignore initialization in this paper.) When a facet calls another facet that resides in another process, whether the called facet is in the same or a different object, the calling thread is suspended. The calling facet accepts the next thread in its queue. Meanwhile, the process sends a message to the other process, indicating (facet, slot), parameters, return point and some identification of the calling object, needed to select the appropriate binding table as explained in the previous section. It is never necessary to send (object, entry), because every object's interface map is replicated in every process, so this can be resolved to (facet, slot) locally. The message is appended to the receiving process's queue.

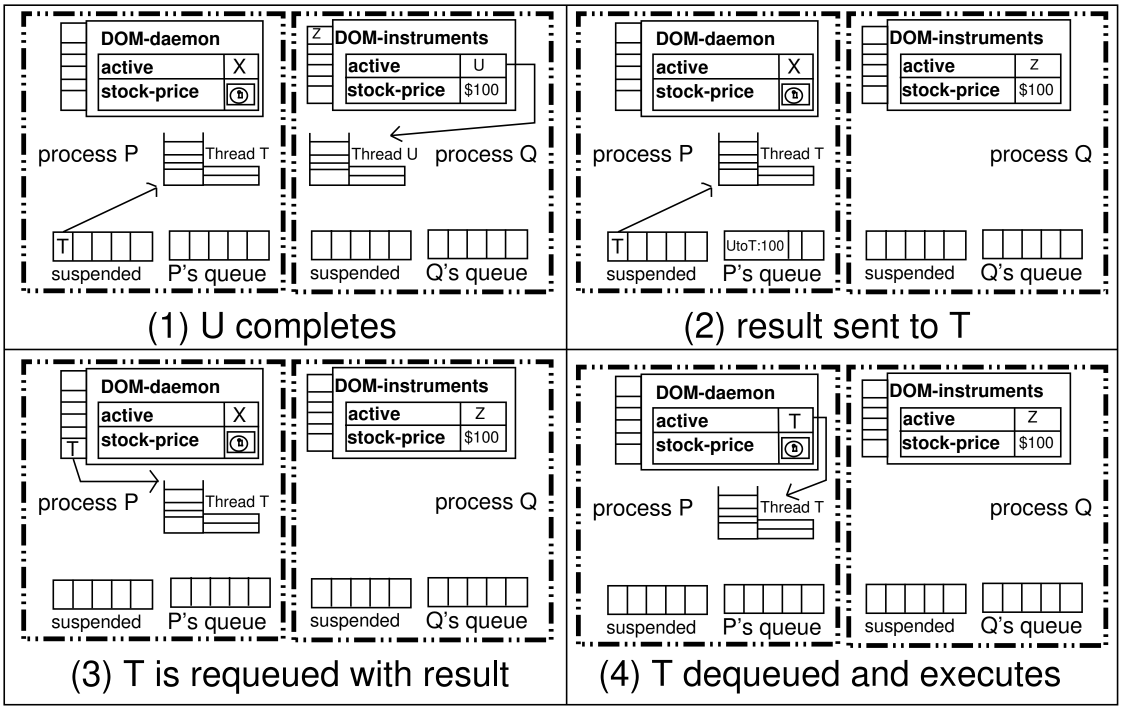

Figure 6-4: Return from Interprocess Call

The receiving process removes the first message in its queue, creates a corresponding thread and queues the thread for the indicated facet. When the call is completed, the process sends a message back to the sending process's queue. The sending process matches the message with the suspended calling thread, and queues the thread for the original calling facet. Figures 6-3 and 6-4 show an example of interprocess call and return, respectively, where the DOM-instruments facet resides in one process while the DOM-daemon facet lives in another. The DOM-daemon calls (polls) the DOM-instruments to get the latest prices.

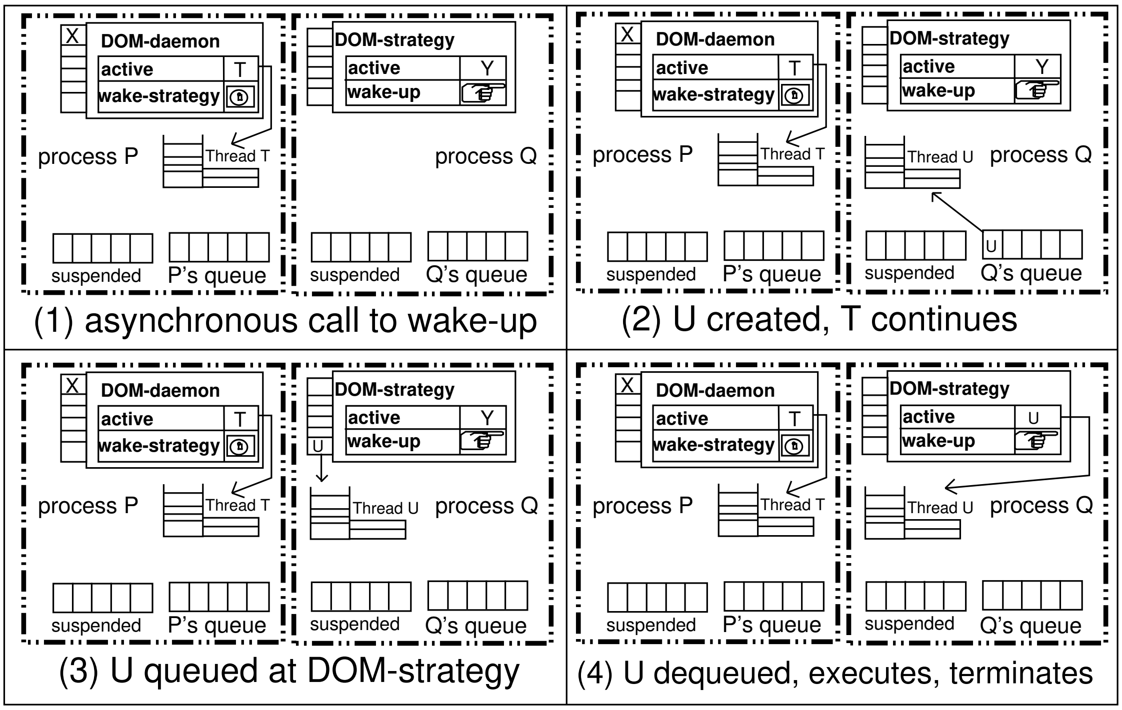

Figure 6-5: DOM-daemon Call to DOM-strategy

Asynchronous calls work similarly to synchronous calls as explained above, with three significant differences. The first is that the process creates a new thread to represent the asynchronous call. The second difference is that the calling facet does not suspend its thread, but instead continues execution. The third is that there is no return point; when an asynchronous call completes, its thread is terminated.[16] Figure 6-5 shows an example where the DOM-daemon facet makes an asynchronous call to notify the DOM-strategy facet of a significant price change (determined from polling, similar to Figure 6-2). Note that this does not permit the calling facet to wait for the completion of the asynchronous call (or, more generally, wait for an asynchronous call from another facet).

It is undesirable to add a receive statement to the PROFIT language, in order to provide some means for blocking an active facet until an asynchronous call, because the waiting time could be extreme. This avoidance of blocking also reveals a problem analogous to the nested monitors problem [Lister 77, Joseph 78]. PROFIT does not provide any way to make an *atomic* call. When a facet makes a synchronous call to another facet, it relinquishes the calling thread and serves the

[16]An optimization is to return the thread to a pool for later reuse, rather than deallocating it.

next thread in its queue. This context switching may result in arbitrary changes to the facet's data slots before the original thread is resumed. In those cases where it is necessary for certain data items to maintain their current values during a synchronous call to another facet, this data must be stored in the local variables of the calling procedure (C function) — and thus in the thread's stack, where there is no danger of being overwritten by unrelated threads queued for the same facet. This lack of atomicity is intentional, since atomicity directly conflicts with the need for rapid reaction to change: an ‘‘atomic’’ call that can be interrupted (and not rolled back), say to meet timing constraints, is of course not really atomic. The various timing and synchronization concerns are elaborated in the next section.

7. Timing and Synchronization

In an ideal world with infinite computation and communication resources, there would be no reason to control how processor time is allocated among threads or how frequently data is polled or messages are sent. But in the real world, computation and communication have real economic costs, and it is necessary to strike a balance between the desired perfect *common knowledge* [Halpern 84] about the current values of rapidly changing data and the pragmatics of shared resources.

In today's computing environment, resources can generally be split between privately used resources and publicly offered services. In the case of privately used resources, the owner must determine how best to allocate them to solve a problem. In the case of publicly offered resources, there are a number of users contending and allocation must be based on some criteria (e.g., equitable allocation, charging per access, nepotism). In our SEC example, the prices database might be a service provided on a mainframe and each portfolio manager might execute on an analyst's workstation. Each analyst would decide how to allocate the computation resources of his workstation to best solve his problem (e.g., how much processor time to allot to a particular daemon or portfolio manager). But the analysts would share communication resources on the network and computation resources on the mainframe, and use would be controlled to prevent degraded service.

There are many approaches to controlling use of resources. Due to the nature of our application, we have to be particularly concerned with timing constraints. Consider again the model illustrated in Figure 2-5, with a logical organization of portfolio manager computations, daemons

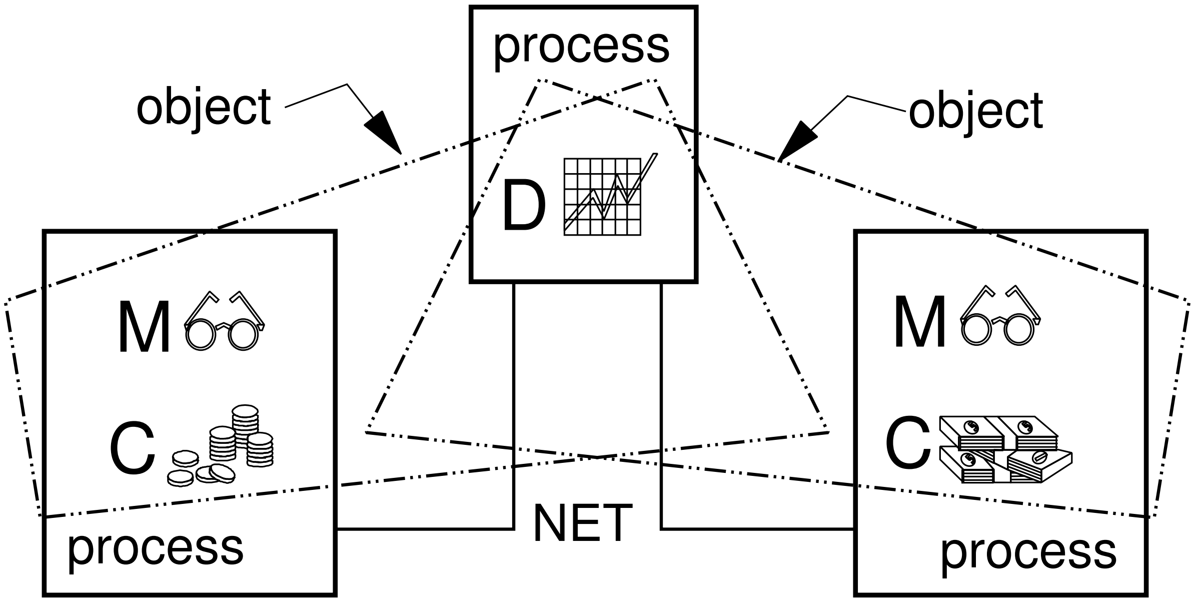

Figure 7-1: Run-Time Organization of SEC with Polling over Network

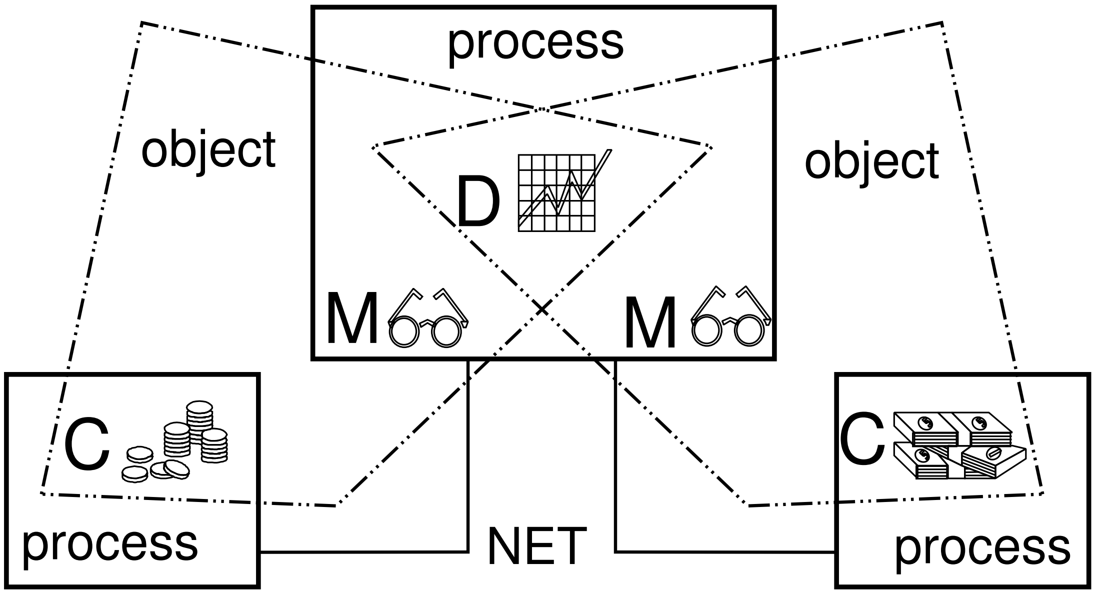

Figure 7-2: Run-Time Organization of SEC with Local Polling

and shared prices. This can be physically organized in several ways. One possibility is shown in Figure 7-1, where the portfolio manager computation and the corresponding daemon reside on the analyst's workstation and the shared data resides on the mainframe. In this case the daemon monitors the prices by polling over the network, and when there is an interesting price change, it interrupts the local computation thread. Another, given in Figure 7-2, places the daemon together with the prices on the mainframe and leaves the portfolio manager computation on the workstation. In this case the daemon monitors by local polling, and when there is an interesting change, sends a message over the network to interrupt the computation. In the former case, the major use of public services is network traffic while in the latter it is processing time on the shared machine.

In either of these cases, the daemon drives the costs. Hence, the PROFIT programmer must be able to control the frequency of these expensive operations. So our first major concern in implementing our application is the timing of repeated operations (i.e., polling). There will be a number of requests from multiple daemons for the same instrument's data, and they will have to be serviced in some order; there are also the updates from the stock exchange wire. It is clear that the updates are more important than accesses with respect to the accuracy of the system, and in fact a daemon most likely would not want to go ahead with its request if it knew that it would return old data. So our second major concern is with priority of different agents.

Finally, after a daemon has received a sequence of prices and noticed an interesting change, it must inform the associated portfolio manager computation. If the manager is idle, there is no problem; this is just a call. However, the manager may be carrying out a time consuming analysis based on the previous prices. If the new information invalidates the analysis, it should be possible to terminate this computation early and alert the manager to the change. So our third major concern is with interrupts.

Thus, in order to support programming of daemons, it is necessary to provide explicit language constructs for stating the timing interval between polls. The simplest construct for this purpose is an `everytime` statement, which executes a given sequence of statements every N time units, where N is a constant. We assume that polling less often costs less than polling more often, where costs may include actual charges for resources and information. We do not intend the `everytime` statement to represent a hard real-time constraint, but assume the system will make ''best effort'' to meet the constraint. If it cannot, it will degrade gracefully and provide proportional service to all competing statements with the highest priority. (A full implementation of PROFIT would include additional, more complex timing statements as in, e.g., [Barbacci 90].)

Because it will not always be possible to meet all timing constraints, it is necessary to associate priorities with different execution components of the program. We have identified three possible places to attach priorities: threads, calls and entries (object interface entries or facet slots). In the threads case, everything executed by the thread would carry its priority. In the calls case, the thread would request a particular priority for the call at the time the call is made. Finally, for entries, any thread that calls a particular procedure would take on the priority of that procedure for the duration of the call. Since one aspect of PROFIT's philosophy is that every facet be self-contained, we chose the third approach, where the priority scheme of one facet need not be coordinated with any other. In the case of a prices facet, for instance, the `put` operation would have a higher priority than the `get` operation because the implementor of the facet determined that `put` is more important for the application. Of course, only the prices feed should actually be permitted to make the `put` call, but we do not address access control and security here.

An `everytime` statement calling the `get` slot would be used in our daemon architecture to poll a shared prices facet. All `get` calls would have lower priority than `put`. But there might be multiple `get` slots (`getgreen`, `getgold`, `getplatinum`, etc.) that the prices facet would use to distinguish multiple classes of daemon clients (say those who have paid a premium for faster service versus those who have not). In this case, these three entries could share a common implementation (using indirection), since the only distinction is the associated priority. In any case, after a daemon has completed a `get` call and noticed that an interesting change has occurred, it is necessary for the daemon to notify the associated computation, that is, the main-line portfolio management strategy.

There are two main ways for this strategy to operate. One is as an infinite loop that repeatedly makes buy and sell decisions based on the current values of prices information and trends in changes to these values. This requires some way to interrupt the loop, in order to be able to change the flow of control so that the strategy notices new values and takes correct action. Another way for the strategy to operate is event-driven, where the portfolio manager does nothing until signalled by a daemon that an important change has occurred and it is necessary to make a decision. If this computation may be longer than the time that would elapse before another change occurs, it is again necessary to be able to interrupt the computation in order to be able to change course in light of new information.

This means an interrupt mechanism is needed. It is not appropriate, however, to be able to interrupt at arbitrary control points, because then the internal data (local variables of procedures as well as data slots) may be left in an inconsistent state. Instead, it must be possible to specify sequences of control points during which interrupts are (or are not) allowed. A common control construct for this purpose is the critical section, within which interrupts are disabled; the classical transaction [Bernstein 87] is similar, in the sense that the system guarantees that the transaction appears to make isolated use of the database. Outside a critical section (or outside a transaction), there are no guarantees about interrupts to control flow (or changes to shared data).

In our application domain, it is relatively rare that a computation will be in a consistent state and therefore there are few points where interrupts should be allowed. Thus, instead of explicitly supporting interrupts, PROFIT provides a special `pause` statement. At every point where a procedure is willing to be interrupted, it executes the `pause` statement. A special ‘‘sensitive section’’ allows the procedure to be interrupted between any two statements within the block of code, to avoid tedious repetition of the `pause` statement. (Note that a sensitive section is the opposite of a critical section, where no interrupts are permitted between instructions.)

The `pause` statement (whether explicit, or implicit in a sensitive section) works by checking the queue of suspended threads. If there is a thread waiting to evaluate a higher priority slot, then the current thread is queued and the waiting thread takes over; thus the facet’s queue must be organized as a priority queue, rather than strictly FIFO. As in an indirect call, a pausing thread has the responsibility to make sure the data in the facet is consistent and any data it needs from the facet is stored within the thread’s stack (that is, the local variables of the in-progress C functions). Allowing pauses only for higher priority operations is important not just for meeting timing constraints, but for suitable allocation of computation resources. For example, a portfolio might want to pause frequently for input from a highly valued ‘‘financial newsletter’’, which might be implemented as a daemon facet shared among multiple portfolio objects, but only look at less valued information when otherwise idle.

Besides interrupts, another alternative to PROFIT’s `pause` statement (and sensitive sections) is rendezvous, mentioned in the previous section in the form of a receive statement. The problem with rendezvous is it implies a forced block, when what we want is to define the opportunities for pre-emption — basically an optional block, where the option is taken only if an appropriate thread is waiting. Explicit blocking in the sense of rendezvous is not acceptable in the context of timing constraints [Baker 87].

8. Implementation

Two subsets of PROFIT, called Asset and Bankrupt, have been implemented. Example stock trading programs written in the two dialects are given in appendices. Each uses a different syntax, as we intentionally avoided defining any particular syntax, and instead concentrated on the underlying programming model. Thus the syntaxes used for the implementations were developed for ease of translation and are not particularly ‘‘user friendly’’.

The Asset implementation was developed first. It translates PROFIT programs into C using `lex` and `yacc`. The resulting programs execute on Sun 3s. Asset supports multiple facets, multiple objects, binding tables, shared facets, and synchronous calls. However, everything executes within a single process, and a simple round-robin scheduler is used. Neither interrupts nor time-critical statements are implemented, and asynchronous calls do not work. There is little error checking or code optimization. Asset consists of two parts, a parser and a run-time kernel that supports scheduling and communication among facets. The parser consists of about 200 lines of `lex` input, 525 lines of `yacc` input, and 2600 lines of C code (not including `lex` and `yacc` output). The kernel consists of about 925 lines of C code.

An Asset program includes facet, object and process declarations as well as C code implementing the facets’ procedure slots. A facet consists of a name, a set of `internal` slots implemented in the facet declaration and a set of `external` slots corresponding to the ***indirect*** slots discussed previously. An object consists of a name, a list of member facets, a set of interface entries and their corresponding facet slots, a binding table mapping facet,slot pairs to facet,slot or object,entry pairs, and a priority specification. The one process has a name, a list of facets, and selects an object,facet,slot triple as the starting point for execution.

The Bankrupt implementation translates PROFIT programs to C++ using `lex`. The SunOS 4.0 lightweight processes library is used, and programs execute correctly on Sun 4s. Strangely, different results are obtained when running the same programs compiled for Sun 3s, and we suspect a bug in the threads implementation. Bankrupt supports an even smaller subset of PROFIT, but gave us an opportunity to experiment with representing facets as C++ objects and using real threads. Bankrupt provides only a parser, with no run-time kernel; the scheduling and communication must be written by the application programmer as part of the facet code (i.e., the C functions), where monitors and threads are employed. The `lex` file consists of about 1000 lines of rules and C++ code.

A Bankrupt program includes C code prefixed by the `C_CODE` keyword, potentially shared facet declarations prefixed by `sh_facet`, object declarations prefixed by `object` and a main program indicated by `profit_main`. Since there is only one process, it is left implicit in this implementation of PROFIT. A facet, shared or otherwise (keyword `facet`), consists of typed data slots and C functions. An object consists of a distinguished facet that defines its interface entries and any non-shared facets. Inclusion of facets in objects and binding of facets to other facets is accomplished through C++'s inheritance mechanism. The main program calls any initialization functions needed and starts the process' threads.

The Asset system is currently being redesigned, based on our Bankrupt system experience, to produce the Capital system. Capital will support asynchronous calls using the lightweight processes package. Special memory management and other optimizations are also being added. The next step will be to support multiple processes on the same machine, and later on multiple machines.

9. Related Work

Among object systems described in the literature, PROFIT is probably closest to SOS [Shapiro 89]. SOS supports a notion of elementary object similar to PROFIT's facets. Elementary objects are collected into contexts (processes) for execution and into groups (objects) to determine allowable communications. A group may contain elementary objects executing in different contexts. However, an elementary object cannot be contained in multiple groups, so there is no sharing. SOS is not intended to address the problems of sharing rapidly changing data.

PROFIT facets are similar to Argus objects [Liskov 88]. If PROFIT objects and processes were not distinct, but instead every object was also a process, the result would closely match Argus guardians. Argus's lightweight processes are analogous to PROFIT's threads. A guardian consists of three parts, an interface (a set of handlers), internal processes and internal state (the objects), which all must reside at the same location. Internal processes and state are hidden within the guardian and accessible only through handlers. When an external call is made to a handler, a process is created within the guardian to service that call. The internal state can be shared among processes within a guardian, but cannot be shared among multiple guardians. Hermes [Bacon 89] is another language that provides both objects and processes. In Hermes, the process defines the external interface, in terms of typed ports; objects are encapsulated inside processes and there is only one thread of control per process. Hermes objects are thus similar to PROFIT facets, except they cannot be shared.

It is interesting to compare PROFIT to Linda [Carriero 89], which defines a paradigm for parallel programming quite different from Argus or Hermes. Linda is probably the best known coordination language: like PROFIT, it extends some base computational language to support communication and coordination among concurrent processes. Communication in Linda is via tuples in a global tuple space. Facets can be viewed as the static analog of tuples. Tuples can be created and deleted dynamically and new processes may be spawned by creating active tuples, while facets cannot be created and deleted during execution and may be (potentially) active or (permanently) inactive depending on whether or not any procedure slots are included in the facet. Linda tuples are shared among all ‘‘interested’’ processes, just as facets are shared among all ‘‘interested’’ objects, that is, their statically defined enclosing objects.

On the other hand, facets may be treated as the dynamic analog of Flavors’s mixins [Moon 86]. Mixins provide data and procedure slots that may be inherited by arbitrary objects. Such inheritance is concerned with structure, however, not contents. In particular, each inheriting object may have different values in the data slots inherited from the same mixin. As mentioned previously, the Jade programming environment [Raj 89] supports a compositional model of object construction, by interfacing large or small program components together. Again, only implementation (code and data definition) is shared, not data values. Facets, in contrast, provide direct sharing of data slot values as well as definitions, in the style of Self [Ungar 87].

In both Self and PROFIT, each slot contains either a data value or procedure code. In Self, all accesses to slots are through messages. When an object receives a message, it is searched for a slot matching the name of the message; if none is found, then the object’s parents are searched recursively (using multiple inheritance rules). Once a match is found, the corresponding slot is evaluated and the result returned. In contrast, all slots referenced within a PROFIT facet must be explicitly declared within that facet, and indirect slots are resolved using a binding table. Thus Self’s sharing is implicit, while PROFIT’s is explicit. PROFIT also has a second sharing mechanism, where a PROFIT object shares data and code with other objects by sharing facets. The corresponding notion in Self would be two objects sharing a common parent, but the analogy is not exact because of the lack of self pointers in PROFIT.

PROFIT is based to some extent on one author's previous experience developing Meld [Kaiser 89], a concurrent object-oriented programming language similar in scope to Avalon [Detlefs 88]. Both Meld and Avalon are particularly concerned with linguistic constructs for transactions and their integration into an object-oriented language, but Meld focuses on concurrency control [Kaiser 90] and Avalon on fault tolerance [Clamen 90]. PROFIT takes an entirely different approach to data sharing in order to support rapid change, which is not addressed by the atomic and serializable transaction model [Eswarnan 76].

Meld supports both compile-time and execution-time collections of objects, with Meld objects akin to PROFIT facets, but using a single grouping construct called the ‘‘feature’’ [Kaiser 87]. Features bundle together related classes and objects and define an interface to other features. One or more features execute within an operating system process. Objects residing in a feature may communicate with remote objects (through a distributed name server) as well as among themselves. Emerald [Black 86] also supports private local objects and shared remote objects, with remote invocations. Like Meld, it also supports intra-object concurrency via multiple operation invocations within an object. While Meld controls concurrent access to private data within objects through transactions (and/or using a simpler construct, called atomic blocks, which simply locks an entire object through the block of code), Emerald uses monitors and condition objects. PROFIT does not allow intra-facet concurrency, since only one thread may execute at a time, but does permit a form of interleaving since another thread can take over while the first is suspended due to a procedure call to another facet or because it interrupts with higher priority. Thus facet consistency is a serious problem, but such interleaving cannot be avoided in the context of real-time constraints.

In the database field, there is a related concept called ‘‘composite objects’’. A composite object is a collection of objects related to each other by the IS-PART-OF relationship. As such, it represents the unit of versioning, physical clustering and concurrency control and authorization [Kim 89]. Composite objects were implemented in Orion [Kim 87].

10. Conclusions

The primary contribution of this paper is our new object model for shared data (facets) based on the separation of compile-time information hiding (objects) and execution-time computation concerns (processes). The sharing is transparent, symmetric, and tightly coupling, our goal presented in the Introduction. We have demonstrated by our financial service example a methodology for using our new object model in a practical application domain with specialized requirements.

One important question raised by this research is the meaning of *correctness* in the context of rapidly changing data. The classical transaction model is clearly inappropriate. Possible approaches that we have considered include modular verification [Hailpern 80, Nguyen 85], self-stabilizing systems [Dijkstra 74, Kessels 88], and common knowledge [Halpern 84]. Using the modular verification technique, the PROFIT programmer would define a set of invariants (defining safety properties) and commitments (defining liveness properties) for each facet. Invariants would be true whenever a facet is not executing, and hence would have to be true whenever a thread leaves a facet (by terminating or by making an inter-facet call) or can be interrupted (e.g., in a sensitive section). Commitments would have to be guaranteed to occur after an operation is invoked. The correctness of objects would then be derived from the conjunction of the invariants and commitments of their constituent facets. One particularly useful kind of invariant is monotonic (or stable) properties: once a monotonic property becomes true, it remains true forever, even though the component variables might change [Hailpern 82, Chandy 86]. These invariants facilitate reasoning about changing histories of entities (facets, objects, or processes), even when one entity is not in direct communication with another.

The problem with this modular-verification approach is the actual statement of what a correctly operating entity really achieves, given the frequently changing data. Trivial statements, such as ‘‘if the price ever gets above X then eventually buy 100 shares’’, would be easy to state but are unlikely to accurately express a desired property, given that prices could change radically before the purchase was complete. The notion that arbitrary data can change during the execution of a piece of code (e.g., evaluation of a procedure slot) suggests an analogy with self-stabilizing systems. In such systems, it is assumed that data associated with an algorithm can be in an ‘‘illegitimate state’’, but that in a finite number of steps it will become stabilized (i.e., in a legitimate state). If these illegitimate states can occur during the life of the algorithm, then they could represent data changing independently of the actions of the algorithm. Hence, to make progress, an algorithm must repeatedly act as if its data has been changed. Correctness of an algorithm starting in an illegitimate state is stated in terms of becoming stabilized in a finite number of steps, hence correctness of the more general case would be stated in terms of eventually reaching a correct answer if enough time passes without any more illegitimate states occurring. In our application, the ‘‘illegitimate states’’ are the transient values of prices, perhaps including wild fluctuations. These states would certainly stop at day-end closing. The severe deficiency of this approach is that a portfolio cannot wait until closing to make buy and sell decisions.

Specifications involving modal logics of knowledge are intended to handle communication of new information, and have proven their effectiveness in describing the computations of distributed systems. In particular, such specifications are useful when reasoning about the actions taken by various entities because of what they ‘‘know’’ about other entities and what these other entities ‘‘know’’. Thus reasoning is not restricted to statements about raw state variables, but, in effect, the history of interactions of each entity with all other entities. Logics of knowledge have demonstrated their usefulness in describing relatively complex interactions with simple, intuitive statements [Halpern 87], but it is not clear how to apply this approach when the knowledge is changing more rapidly than it can be sampled.

Formal semantics along with good specification, verification and validation techniques for PROFIT in particular and our object model in general appear to be an exciting area for future research.

Acknowledgments

Isai Shenker developed the Asset implementation of PROFIT, and Ari Gross is responsible for Bankrupt. Tushar Patel, Jason Kim, Michael Mayer and Vanessa Cole have worked on Capital. Steve Popovich collaborated with the first author on an earlier notion of facets [Popovich 88] intended as an extension to the Meld language. We would like to thank Kim Gostelow, Ken Perry, Tom Reps and Dave Ungar for useful discussions, and Tien Huynh and Catherine Lassez for motivating the financial services example [Lassez 90]. Nasser Barghouti, Wenwey Hseush, Peter Wegner and Travis Winfrey made extensive comments on previous drafts of this paper. We would also like to thank the anonymous referees for their helpful suggestions.

An extended abstract of this work was presented at the International Conference on Computer Languages, New Orleans LA, March 1990, under the title *An Object Model for Shared Data*. Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, Citibank, DEC, IBM, Siemens, SRA, Sun and

Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

References

[Agha 86] Gul Agha.
Actors A Model of Concurrent Computation in Distributed Systems.
The MIT Press, Cambridge MA, 1986.

[Bach 86] Maurice J. Bach.
The Design of the UNIX Operating System.
Prentice-Hall, Englewood Cliffs NJ, 1986.

[Bacon 89] David F. Bacon and Robert E. Strom.
Implementing the Hermes Process Model.
Technical Report RC 14518, IBM T.J. Watson Research Center, March, 1989.

[Baker 87] T.P. Baker.
A Low-Level Tasking Package for Ada.
In *Using Ada: ACM SIGAda International Conference*, pages 141-146. December, 1987.
Special issue of *Ada LETTERS*, December 1987.

[Barbacci 90] Mario R. Barbacci and Jeannette M. Wing.
A Language for Distributed Applications.
In *International Conference on Computer Languages*, pages 59-68. New Orleans LA, March, 1990.

[Bernstein 87] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, Reading MA, 1987.

[Birrell 84] A.D. Birrell and B.J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[Black 86] Andrew Black, Norman Hutchinson, Eril Jul and Henry Levy.
Object Structure in the Emerald System.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 78-86. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Carriero 89] Nicholas Carriero and David Gelernter.
Linda in Context.
Communications of the ACM 32(4):444-458, April, 1989.

[Chambers 89] Craig Chambers, David Ungar and Elgin Lee.
An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 49-70. ACM Press, New Orleans LA, October, 1989.
Special issue of *SIGPLAN Notices*, 24(10), October 1989.

[Chandy 86] Mani Chandy and Jayadev Misra.
An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection.
ACM Transactions on Programming Languages and Systems 8(3):326-343, July, 1986.

[Charniak 80] Eugene Charniak, Christopher K. Riesbeck and Drew V. McDermott.
Artificial Intelligence Programming.
Lawrence Erlbaum Assoc., Hilsdale NJ, 1980.

[Ciancarini 90] Paola Ciancarini.
Coordination Languages for Open System Design.
In *International Conference on Computer Languages*, pages 252-260. New Orleans LA, March, 1990.

[Clamen 90] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles and Jeannette M. Wing.
Reliable Distributed Computing with Avalon/Common Lisp.
In *International Conference on Computer Languages*, pages 169-179. New Orleans LA, March, 1990.

[Coffin 89] Michael H. Coffin and Gregory R. Andrews.
Towards Architecture-Independent Parallel Programming.
Technical Report TR 89-21a, The University of Arizona, September, 1989.

[Cooper 88] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Carnegie Mellon University Department of Computer Science, June, 1988.

[Dasgupta 88] Partha Dasgupta, Richard J. Leblanc Jr. and William F. Appelbe.
The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work.
In *8th International Conference on Distributed Computing Systems*, pages 2-9. San Jose CA, June, 1988.

[Detlefs 88] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer 21(12):57-69, December, 1988.

[Dijkstra 74] Edsger W. Dijkstra.
Self-stabilizing Systems in Spite of Distributed Control.
Communications of the ACM 17(11):643-644, November, 1974.

[Dittrich 86] Klaus Dittrich and Umeshwar Dayal (editors).
1986 International Workshop on Object-Oriented Database Systems.
IEEE Computer Society Press, Pacific Grove CA, 1986.

[Eswarnan 76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[Faulk 88] Stuart R. Faulk and David L. Parnas.
On Synchronization in Hard-Real-Time Systems.
Communications of the ACM 31(3):274-287, March, 1988.

[Gibbons 87] Phillip B. Gibbons.
A Stub Generator for Multilanguage RPC in Heterogenous Environments.
IEEE Transactions on Software Engineering SE-13(1):77-87, January, 1987.

[Goldberg 83] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley, Reading MA, 1983.

[Haeberli 88] Paul E. Haeberli.
ConMan: A Visual Programming Language for Interactive Graphics.
In *SIGGRAPH '88*, pages 103-111. Atlanta GA, August, 1988.
Special issue of *Computer Graphics*, 22(4), August 1988.

[Hailpern 80] Brent T. Hailpern.
Verifying Concurrent Processes Using Temporal Logic.
PhD thesis, Stanford University, September, 1980.
Available as *Lecture Notes in Computer Science 129*, Springer-Verlag, 1982.

[Hailpern 82] Brent Hailpern and Susan Owicki.
Modular Verification of Concurrent Programs.
In *9th ACM Symposium on Principles of Programming Languages*, pages 322-336. Albuquerque NM, January, 1982.

[Hailpern 90] Brent Hailpern and Gail E. Kaiser.
Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data.
Technical Report RC 16232, IBM Research Division, October, 1990.
Submitted for publication.

[Halpern 84] J. Halpern and Y. Moses.
Knowledge and Common Knowledge in a Distributed Environment.
In *3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50-61. Vancouver BC, Canada, August, 1984.

[Halpern 87] Joseph Y. Halpern and Lenore D. Zuck.
A Little Knowledge Goes a Long Way: Simple knowledge-based derivations and correctness proofs for a family of protocols.
In *6th Annual ACM Symposium on Principles of Distributed Computing*, pages 269-280. Vancouver BC, Canada, August, 1987.

[Joseph 78] M. Joseph and V. R. Prasad.
More on Nested Monitor Calls.
Operating Systems Review 12(2):21-25, 4, 1978.

[Kaiser 87] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
IEEE Software 4(4):17-24, July, 1987.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
MELDing Multiple Granularities of Parallelism.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 147-166. Cambridge University Press, Nottingham, UK, July, 1989.

[Kaiser 90] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich and Shyhtsun F. Wu.
Multiple Concurrency Control Policies in an Object-Oriented Programming System.
In *Second IEEE Symposium on Parallel and Distributed Processing*, pages 623-626. Dallas TX, December, 1990.

[Kessels 88] J.L.W. Kessels.
An Exercise In Proving Self-Stabilization With A Variant Function.
Information Processing Letters (29):39-42, 15 September, 1988.

[Kim 87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garz and Darrel Woelk (editor).
Composite Object Support in an Object-Oriented Database System.
ACM Press, 1987.
Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[Kim 89] Won Kim, Elisa Bertino and Jorge F. Garza.
Composite Objects Revisited.
In James Clifford, Bruce Lindsay and David Maier (editors), *1989 ACM SIGMOD International Conference on the Management of Data*, pages 337-437. ACM Press, Portland OR, June, 1989.
Special issue of *SIGMOD Record*, 18(2), June 1989.

[Lassez 90] Catherine Lassez and Tien Huynh.
An Expert Decision-Support System for Option-Based Investments Strategies.
Journal of Computers and Mathematics with Applications , 1990.
In press.

[Li 86] Kai Li and Paul Hudak.
Memory Coherence in Shared Virtual Memory Systems.
In *5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229-239. Calgary Alberta, Canada, August, 1986.

[Lieberman 86] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 214-223. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Liskov 88] Barbara Liskov.
Distributed Programming in Argus.
Communications of the ACM 31(3):300-312, March, 1988.

[Lister 77] Andrew Lister.
The Problem of Nested Monitor Calls.
Operating Systems Review 11(2):5-7, July, 1977.

[Mazumdar 89] Subrata Mazumdar and Aurel A. Lazar.
Knowledge-Based Monitoring of Integrated Networks.
In Branislav Meandzija and Jil Westcott (editors), *IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, pages 235-243. North-Holland, Boston MA, May, 1989.

[Moon 86] David A. Moon.
Object-Oriented Programming with Flavors.
In Norman Meyrowitz (editor), *Object-Oriented Systems, Languages, and Applications Conference*, pages 1-8. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Nguyen 85] Van Nguyen, David Gries and Susan Owicki.
A Model and Temporal Proof System for Networks of Processes.
In *12th ACM Symposium on Principles of Programming Languages*, pages 121-131. New Orleans LA, January, 1985.

[Peinl 88] Peter Peinl, Andrea Reuter and Harald Sammer.
High Contention in a Stock Trading Database: A Case Study.
In *ACM SIGMOD International Conference on the Management of Data*, pages 260-268. Chicago IL, June, 1988.
Special issue of *SIGMOD Record*, 17(3), September 1988.

[Popovich 88] Steven S. Popovich and Gail E. Kaiser.
Melding Parallel and Distributed Programming.
Technical Report CUCS-402-88, Columbia University Department of Computer Science, October, 1988.

[Raj 89] Rajendra K. Raj and Henry M. Levy.
A Compositional Model for Software Reuse.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 3-24. Cambridge University Press, Nottingham, UK, July, 1989.

[Rashid 87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31-39. Palo Alto CA, October, 1987.
Special issue of *SIGPLAN Notices*, 22(10), October 1987.

[Shapiro 89] Marc Shapiro, Philippe Gautron and Laurence Mosseri.
Persistence and Migration for C++ Objects.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 191-204. Cambridge University Press, Nottingham, UK, July, 1989.

[Stefik 86] Mark J. Stefik, Daniel G. Bobrow and Kenneth M. Kahn.
Integrating Access-Oriented Programming into a Multiparadigm Environment.
IEEE Software 3(1):11-18, January, 1986.

[Stein 89] Lynn Andrea Stein, Henry Lieberman and David Ungar.
A Shared View of Sharing: The Treaty of Orlando.
Object-Oriented Concepts, Databases, and Applications.
ACM Press, New York, 1989, pages 31-48.

[Strom 84] Robert Strom and Nagui Halim.
A New Programming Methodology for Long-Lived Software Systems.
IBM Journal of Research and Development 28(1):52-59, January, 1984.

[Ungar 87] David Ungar and Randall B. Smith.
Self: The Power of Simplicity.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 227-242. ACM Press, Orlando FL, October, 1987.
Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[Wedde 89] Horst F. Wedde, Bogden Korel, Willie G. Brown and Shengdong Chen.
Transparent Distributed Object Management Under Completely Decentralized Control.
1989.
Wayne State University.

[Wegner 87] Peter Wegner.
Dimensions of Object-Based Language Design.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 168-182. ACM Press, Orlando FL, October, 1987.
Special issue of *SIGPLAN Notices*, 22(12), December 1987.

I. Example Using Asset Dialect

This example interacts with the end-user and gives advice based on rules defined in the PROFIT program. The end-user decides whether to buy or sell AT&T and IBM stocks. This program was written by Isai Shenker.

```
{*
        Declaration of Profit facets, objects, and processes
*}

{*              Stock A facet           *}
FACET stock_a
internal init_a(): int,         {*initialization*}
        low: int,               {*lowest price*}
        high: int,              {*highest price*}
        ave: int,               {*average price*}
        number: int,            {*number of shares*}
        ret_num(): int,         {*utility*}
        name: char *,           {*stock name*}
        get_ok(): int,          {*selling policy*}

{*
        The following are accessor functions serving as
        corresponding object entries
*}
        get_name(): char *


external delta_b: int           {*price change of stock B*}
END FACET stock_a


{*              Stock B facet           *}

Facet stock_b
internal init_b(): int,         {*initialization*}
        sell(): int,            {*selling action*}
        b_name: char *,         {*stock name*}
        price_change: int,      {*latest price change*}
        quality: int,           {*international rating*}
        number_b: int,          {*number of shares*}

{*
        The following are accessor functions serving as
        corresponding object entries
*}
        get_b_name(): char *

external a_number(): int,       {*number of shares of stock A*}
        a_name: char *          {*stock A name*}

end facet stock_b


{*      Securities and Exchange Committee (SEC) facet *}

Facet sec
internal init_sec(): int,       {*initialization*}
        get_number(): int,      {*total number of shares*}
        report(): int,          {*display SEC data*}
        get_most(): int         {*find the most popular stock*}

external number_a(): int,       {*number of shares of stock A*}
        a_name: char *,         {*name of stock A*}
        a_low: int,             {*lowest price of stock A*}
        a_high: int,            {*highest price of stock A*}
        foo: int,
        b_name: char *,         {*name of stock B*}
        b_change: int,          {*price change of stock B*}
        number2: int            {*number of shares of stock B*}
end facet sec


{*      Broker facet            *}

Facet broker
internal sell_a(): int,
        sell_b(): int,
        buy_a(): int,
        buy_b(): int,
        advise(): int,
        init_broker(): int      {*initialization*}

external low_a: int,
        a_name: char *,
        b_name: char *,
        high_a: int,
        number_a: long,
        number_b: long,
        ok_to_sell_a(): int,
        delta_b: int,
        total_number(): long,
        sec_query(): int,
        go_sell_b(): int
end facet broker


{*Main Driver for New York Stock Exchange -- orders are accepted by this facet*}

Facet initial
internal first(): int

external init_A(): int,
        init_B(): int,
        init_SEC(): int,
        init_BROKER(): int,
        get_report(): int,
        get_advise(): int,
        action_buy_a(): int,
        action_buy_b(): int,
        action_sell_a(): int,
        action_sell_b(): int
end facet initial

OBJECT stocks
        FACETS: stock_a stock_b

        ENTRY:
        {*Mapping of object entries to procedure slots*}

        stocks_a_get_ok -> stock_a.get_ok,
        stocks_init_a -> stock_a.init_a,
        stocks_init_b -> stock_b.init_b,
        stocks_b_sell -> stock_b.sell,
        stocks_a_number -> stock_a.ret_num;

        MAP:
        {*Mapping of all external slots*}

        stock_a.delta_b -> stock_b.price_change,
        stock_b.a_number -> stock_a.ret_num,
        stock_b.a_name -> stock_a.name;

        PRIORITY:
```

```
        resp_pr := 2;
        inter_pr := 3;
        slot_pr := 2;
END OBJECT stocks


OBJECT monitor
        FACETS: sec stock_a stock_b

        ENTRY:
        {*Mapping of object entries to procedure slots*}

        monitor_most -> sec.get_most,
        monitor_total -> sec.get_number,
        sec_report -> sec.report,
        sec_init_sec -> sec.init_sec;

        MAP:
        {*Mapping of all external slots*}

        sec.number_a -> stocks.stocks_a_number,
        sec.a_name -> stock_a.name,
        sec.b_name -> stock_b.b_name,
        sec.b_change -> stock_b.price_change,
        sec.a_high -> stock_a.high,
        sec.a_low -> stock_a.low,
        sec.foo -> stock_a.ave,
        sec.number2 ->stock_b.number_b,

        stock_a.delta_b -> stock_b.price_change,
        stock_b.a_number -> stock_a.ret_num,
        stock_b.a_name -> stock_a.name;

        PRIORITY:
        resp_pr := 1;
        inter_pr := 0;
        slot_pr := 4;
END OBJECT monitor

OBJECT brokerage
        FACETS: broker stock_a stock_b

        ENTRY:
        {*Mapping of object entries to procedure slots*}

        broker_init -> broker.init_broker,
        broker_advise -> broker.advise,
        broker_sell_a -> broker.sell_a,
        broker_sell_b -> broker.sell_b,
        broker_buy_a -> broker.buy_a,
        broker_buy_b -> broker.buy_b;

        MAP:
        {*Mapping of external slots*}

        stock_a.delta_b -> stock_b.price_change,
        stock_b.a_number -> stock_a.ret_num,
        stock_b.a_name -> stock_a.name,

        broker.low_a -> stock_a.low,
        broker.a_name -> stock_a.name,
        broker.b_name -> stock_b.b_name,
        broker.high_a -> stock_a.high,
        broker.number_a -> stock_a.number,
        broker.number_b -> stock_b.number_b,
        broker.ok_to_sell_a -> stock_a.get_ok,
        broker.sec_query -> monitor.monitor_most,
        broker.go_sell_b -> stocks.stocks_b_sell,
        broker.total_number -> monitor.monitor_total,
        broker.delta_b -> stock_b.price_change,

        PRIORITY:
        resp_pr := 1;
        inter_pr := 0;
        slot_pr := 4;

END OBJECT brokerage


OBJECT system
        FACETS: initial

        ENTRY:

        MAP:
        initial.init_A -> stocks.stocks_init_a,
        initial.init_B -> stocks.stocks_init_b,
        initial.init_SEC -> monitor.sec_init_sec,
        initial.init_BROKER -> brokerage.broker_init,
        initial.get_report -> monitor.sec_report,
        initial.get_advise -> brokerage.broker_advise,
        initial.action_sell_a -> brokerage.broker_sell_a,
        initial.action_sell_b -> brokerage.broker_sell_b,
        initial.action_buy_a -> brokerage.broker_buy_a,
        initial.action_buy_b -> brokerage.broker_buy_b;

        PRIORITY:
        resp_pr := 2;
        inter_pr := 3;
        slot_pr := 2;
END OBJECT system

{*      UNIX process -- home for all facets     *}

PROCESS nyse
        FACETS: stock_a stock_b broker sec initial

        {*first procedure slot to execute*}
        start := system.initial.first();

END PROCESS nyse

#include "stock_a.h"

char *stock_a_name;

int get_ok()
{
        if (delta_b > 20) {
                return(1);
        }
        else {
                return(0);
        }
}


int ret_num()
{
        return(number);
}

int init_a()
{
```

```
        stock_a_name = "AT&T";
        put_slot(_name, stock_a_name);
        put_slot(_low, 24);
        put_slot(_high, 47);
        put_slot(_ave, 40);
        put_slot(_number, 1000);
}

/*
        accessor routines
*/
char    *get_name()
{
        return(name);
}

int     get_high()
{
        return(high);
}

int     get_ave()
{
        return(ave);
}

int     get_low()
{
        return(low);
}

#include "stock_b.h"

char *stock_b_name;

int init_b()
{
        stock_b_name = "IBM";

        put_slot(_b_name, stock_b_name);
        put_slot(_price_change, 10);
        put_slot(_number_b, 1000);
}


int sell()
{

        put_slot(_price_change, price_change + 10);
        if (number_b <= 700)
                printf("nyse update: stock %s is in great demand\n", b_name);

}

/*
        accessor functions
*/

int     get_number_b()
{
        return(number_b);
}

char    *get_b_name()
{
        return(b_name);
}
```

```
int     get_change()
{
        return(price_change);
}

#include "sec.h"

long get_number()
{
long num;

        num = _number_a() + number2;

        return(num);
}


int get_most ()
{

        if (_number_a() > number2) {
                return(1);
        }
        else {
                return(2);
        }
}


int init_sec()
{
}

#include "sec.h"
report()
{
        printf("\n\n\tNew York Stock Exchange Report\n\n");
        printf("\n\tstock_name:\t%s\n", a_name);
        printf ("\tnumber:\t%ld\n", _number_a());
        printf("\thigh = %d\n\tlow = %d\n\taverage = %d",
               a_high, a_low, foo);

        printf("\n\tname:\t%s\n", b_name);
        printf ("\tnumber:\t%ld\n", number2);
        printf("\tprice change: %d\n", b_change);

        printf("\ntotal number of shares is %ld\n", _get_number());
}

#include "broker.h"

#define SELL_A  1
#define SELL_B  2
#define BUY_A   3
#define BUY_B   4

int init_broker()
{
}

int sell_b()
{
int stock;

        printf("request to sell %s\n", b_name);
        upd_nyse(SELL_B);
        stock = _sec_query();
```

```
        if (stock == 1) {
                printf("Cannot sell %s stock at a reasonable price\n", b_name);
                printf("Investors are crazy about %s shares\n", a_name);
                printf("and do not want to buy other Blue Chips\n");
                put_slot(_number_a, number_a - 200);
                if (number_a < 0) put_slot(_number_a, number_a + 300);
        }
        else {
                _go_sell_b();
                printf("Sold %s\n", b_name);
                printf("%s’s price change is %d\n", b_name, delta_b);
        }
}


int sell_a()
{
        printf("request to sell %s\n", a_name);
        upd_nyse(SELL_A);
        if (_ok_to_sell_a()) {
                printf("Sold %s\n", a_name);
                if (delta_b >= high_a - low_a)
                        printf("Stock %s is fluctuating more than stock %s !\n",
                                b_name, a_name);
                else {
                        put_slot(_delta_b, delta_b - 50);
                        printf("Stock %s is too risky, sell it\n", a_name);
                }

        }
        else {

                if (delta_b >= 10) {
                        printf
                ("Cannot sell %s stock -- price of %s is not changing\n",
                                                                a_name, b_name);
                        put_slot(_delta_b, delta_b + 10);
                }
                else {
                        put_slot(_number_a, number_a+30);
                        put_slot(_high_a, high_a+15);
                        put_slot(_delta_b, delta_b + 33);
                        printf("Sold %s\n", a_name);
                }

        }
}


int buy_a()
{
        printf("request to buy %s\n", a_name);
        upd_nyse(BUY_A);
        printf("Bought %s\n", a_name);
        if (_total_number() >= 2005) {
                printf("advise: sell %s -- market is flooded\n", b_name);
                put_slot(_delta_b, delta_b - 39);
        }
        else if (_ok_to_sell_a() == 0)
                printf("advise: buy %s, bargain price at %d\n",
                        a_name, low_a);
        else {
                put_slot(_number_a, number_a - 150);
                if (number_a < 0) put_slot(_number_a, -1*number_a);
        }
}
```

```
int buy_b()
{
        printf("request to buy %s\n", b_name);
        upd_nyse(BUY_B);
        if (high_a > 160) {
                printf("advise: buy %s\n", a_name);
                printf("stock %s’s high was at %d\n", a_name, high_a);
                printf("there are %d shares of this stock\n", number_a);
                printf("total number of shares is %d\n", _total_number());
        }
        else {
                printf("Bought %s \n", b_name);
                if (delta_b > 100)
                        printf("Warning: sell %s ASAP\n", b_name);
                else
                        printf("Don’t sell %s\n", b_name);
        }
}


upd_nyse(action)
int action;
{

        switch (action) {
        case SELL_A:
                put_slot(_low_a, high_a - 60);
                if (low_a < 0) put_slot(_low_a, -1*low_a);

                put_slot(_high_a, high_a - 30);
                if (high_a < 0) put_slot(_high_a, -1*high_a);

                put_slot(_number_a, number_a - 100);
                if (number_a < 0) put_slot(_number_a, -1*number_a + 20);

                put_slot(_number_b, number_b - 30);
                if (number_b < 0) put_slot(_number_b, -1*number_b);

                put_slot(_delta_b, delta_b - 90);
                break;

        case SELL_B:
                put_slot(_high_a, high_a - 20);
                if (high_a < 0) put_slot(_high_a, -1*high_a);

                put_slot(_low_a, high_a);
                put_slot(_delta_b, delta_b + 131);
                put_slot(_number_a, number_a + 250);

                put_slot(_number_b, number_b - 100);
                if (number_b < 0) put_slot(_number_b, -1*number_b + 100);
                break;

        case BUY_A:
                put_slot(_high_a, high_a + 20);

                put_slot(_number_a, number_a - 50);
                if (number_a < 0) put_slot(_number_a, -1*number_a + 41);

                put_slot(_number_b, number_b + 300);
                break;

        case BUY_B:
                put_slot(_low_a, low_a + 25);
                put_slot(_high_a, high_a + 47);
                put_slot(_delta_b, delta_b + 23);
```

```
                put_slot(_number_a, number_a - 135);
                if (number_a < 0) put_slot(_number_a, -1*number_a + 18);
                break;
        }

        if (low_a > high_a)
                put_slot(_high_a, low_a + (int) (low_a - high_a)/3);
}


int advise()
{
int stock;

        stock = _sec_query();

        if (stock == 2) {
                if (number_b >= 800) {
                        if (high_a >= 100) {
                                printf("Buy %s, it's price is rising\n",
                                        b_name);
                                if (delta_b > 40)
                                        printf(
                                "Stock %s has to be monitored, it is fluctuating\n",
                                        b_name);
                        }
                        else {
                                printf("Hold on to %s\n", a_name);
                                printf("Start aggressive investment in %s\n",
                                                                b_name);
                        }
                }
                else {
                        printf("Investors are dumping all stocks\n");
                        printf("A good oportunity to buy %s and %s\n",
                                        a_name, b_name);
                }
                return;
        }

        if (delta_b <= high_a - low_a) {
                if (low_a >=250) {
                        printf("Stock %s is extremely attractive\n", a_name);
                        printf("Invest all money in %s, sell %s\n",
                                                        a_name, b_name);
                }
                else if (low_a >= 150) {
                        printf("Stock %s is recommended\n", a_name);
                        printf("It is a good long-term investment\n");
                        printf("Do not sell %s\n", b_name);
                }
                else {
                        printf("Stock %s is investors' favorite\n", a_name);
                        printf("Buy it as a safe invetment with low returns\n");
                }
        }
        else {
                printf("Stock %s is going down, sell it\n", a_name);
                printf("Do not buy %s\n", b_name);
        }
}

#include "initial.h"

first()
{
int ans;
```

```
        printf("\n\n\tWelcome to the demonstration of PROFIT\n\n");
        _init_A();
        _init_B();
        _init_SEC();
        _init_BROKER();

        while (1) {
                printf("\n\n\n");
                printf
("\t\t1 - sell AT&T\n\t\t2 - buy AT&T\n\t\t3
  - sell IBM\n\t\t4 - buy IBM\n\t\t5 - report\n\t\t6 -
  get inside info\n\t\t7 - exit\n");

                printf("\n\t\tSelection: ");
                scanf ("%d", &ans);
                system("clear");
                switch(ans) {
                case 1:
                        _action_sell_a();
                        break;
                case 2:
                        _action_buy_a();
                        break;
                case 3:
                        _action_sell_b();
                        break;
                case 4:
                        _action_buy_b();
                        break;
                case 5:
                        _get_report();
                        break;
                case 6:
                        _get_advise();
                        break;
                case 7:
                        exit(0);
                        break;
                default:
                        printf("Invalid option\n");
                        break;
                }
        }
}
```

II. Example Using Bankrupt Dialect

This example is a simulated programmed trading system. It automatically buys and sells any number of stocks for any number of clients according to simple rules provided at start-up. This program was written by Ari Gross.

```
C_CODE   #include <math.h>

C_CODE   #define  FLUX             10
C_CODE   #define  MAX_CUSTOMER     4
C_CODE   #define  MAX_STOCKS       4
C_CODE   #define  BUY              1
C_CODE   #define  SELL             2
C_CODE   #define  SET              1
C_CODE   #define  UNSET            0


sh_facet  stocks {
        int  stock_prices[MAX_STOCKS];  //array for storing stock prices
        int  number_of_stock_prices;

        void print_out_stock_prices() {
              int j = number_of_stock_prices;
              for(int i=1; i<j+1; i++)
                  fprintf(stdout, "\nSTOCKS:    %d: %d   ",i, stock_prices[i]);
              fprintf(stdout,"\n"); fflush(stdout);
        }
}; end_sh_facet



sh_facet  demon_1 :  sh_facet  stocks  {

                int     rules[MAX_STOCKS+1][2];
                int     client_list[MAX_STOCKS+1][MAX_CUSTOMER+1];
                int     number_of_rules;
                int     number_of_client_list;
                int     mailbox[MAX_CUSTOMER+1][MAX_STOCKS+1];
                int     demon_init;


void    init()
{
        int  client_id, monitor_num, stockn, p_buy, p_sell;
        int  client_rule;
        if (demon_init==0) {
                demon_init = 1;
                fprintf(stdout, "\nDAEMON_CITI:        demon initialization: ");
                fprintf(stdout,
"\nDAEMON_CITI:        Enter number of stocks to monitor  ");
                fscanf(stdin, "%d", &monitor_num);
                number_of_rules = monitor_num;
                fprintf(stdout, "\nDAEMON_CITI:        number of stocks being monitored:  %d",
get_number_of_rules());
                for (int i=1; i<number_of_rules()+1; ++i) {
                        fprintf(stdout,
"\nDAEMON_CITI:        Enter stock number and buying and selling price  ");
                        fflush(stdout);
                        fscanf(stdin, "%d %d %d", &stockn, &p_buy, &p_sell);
                        rules[stockn][BUY] =  p_buy;
                        rules[stockn][SELL] = p_sell;
                        fprintf(stdout,
"\nDAEMON_CITI:        How many clients are associated with this rule?  ");
                        fscanf(stdin, "%d", &client_rule);
                        if (i==1 || (client_rule > number_of_client_list))
                                number_of_client_list = client_rule;
                        for (int j=1; j<client_rule+1; j++){
                                fprintf(stdout, "\nDAEMON_CITI:        Enter client id: ");
                                fscanf(stdin, "%d", &client_id);
                                client_list[i][j] = client_id;
                        }
                }
        print_out_rules();
        print_out_client_list();
        fprintf(stdout, "initialization of DAEMON_CITI complete\n");
        fflush(stdout);
      }
}

void    run()
{
        int  buy, sell, var_assign;
        for(;;) {
             fprintf(stdout, "\nDAEMON_CITI:        checking stock prices...\n");
             for(int i=1; i<number_of_rules+1; i++) {
                     buy  = rules[i][BUY];
                     sell = rules[i][SELL];
                     if (buy > stock_prices[i]){
                             for(int j=1; j<number_of_client_list()+1;j++)
                                       if (client_list[i][j]!=0)
                                         {    fprintf(stdout,
"\nDAEMON_CITI:      notify client %d to BUY stock %d\n", client_list[i][j], i);
                                              mailbox[client_list[i][j]][i] = BUY;
                                              mailbox[client_list[i][j]][0] = SET;
                                              fprintf(stdout,"%d\n",
                                                mailbox[client_list[i][j]][i]);
                                              fflush(stdout);
                                              fprintf(stdout,
"\nDAEMON_CITI:        stock %d:  buy price:  %d , cur price: %d\n",
i, buy, stock_prices[i]);
                                              fflush(stdout);
                                              }
                     }

                       if (sell < stock_prices[i]){
                             for(int j=1; j<number_of_client_list+1;j++)
                                if (client_list[i][j]!=0)
                                 {
                                     fprintf(stdout,
"\nDAEMON_CITI:        notify client %d to SELL stock %d\n", client_list[i][j], i);
                                     mailbox[client_list[i][j]][i] = SELL;
                                     mailbox[client_list[i][j]][0] = SET;
                                     fprintf(stdout,"%d\n",
                                             mailbox[client_list[i][j]][i]);
                                     fflush(stdout);
                                     fprintf(stdout,"\nDAEMON_CITI:
sell price:  %d , cur price: %d\n", sell, stock_prices[i]);
                                     fflush(stdout);
                                   }
                     }
                   }
           }
      }


     void    print_out_rules()
         {
               for (int i=1; i<number_of_rules+1; ++i)
                 fprintf(stdout,
"\nDAEMON_CITI:        stock: %d  buy: %d  sell: %d\n",
i, get_rules[i][BUY], rules[i][SELL]);
               }

     void    print_out_client_list()
```

```
        {
                for (int i=1; i<number_of_rules()+1; ++i){
                        fprintf(stdout,"\nDAEMON_CITI:        stock: %d  ", i);
                for (int j=1; j<number_of_client_list+1; ++j)
                        fprintf(stdout,"\nDAEMON_CITI:        %d",
                                client_list[i][j])
                        }
                fprintf(stdout, "\n"); fflush(stdout);
               }
}; end_sh_facet




object  sec : sh_facet stocks {
   facet sec {
       interface:

           int     stock_value;

           void    run() {
             int            ran, inc, c_stock, tprice, counter;
             long int       random();
             for(;;) {
                fprintf(stdout, "\nNYSE:          ");
                for(c_stock=1; c_stock<number_of_stock_prices+1; ++c_stock) {
                    inc =  ((int)random()) % (FLUX + 1);
                    inc -= FLUX/2;
                    if (inc<0)
                       fprintf(stdout,"%d:  %d %d    ", c_stock, stock_prices[c_stock], inc);
                    else
                        fprintf(stdout,"%d:  %d +%d    ", c_stock, stock_prices[c_stock], inc);
                    stock_prices[c_stock] = stock_prices[c_stock] + inc;
                }
            }
}

void
init()
{
        int     dvalue, snum;
        void    begin_day();

        begin_day();

        fprintf(stdout, "\nSEC:           Enter number of stocks to be traded: ");
        fscanf(stdin, "%d", &snum);
        number_of_stock_prices = snum;

        for (int i=1;i<number_of_stock_prices+1;i++)  {
          fprintf(stdout, "\nSEC:           Enter price of stock %d: ", i);
          fscanf(stdin,"%d", &stock_value);
          put_stock_prices(i, stock_value); }
          fprintf(stdout,
"\nSEC:           Here are stock prices at the opening of the trading day ... :\n\n");
          print_out_stock_prices();
}

     }; end_facet

}; end_object
```

```
sh_facet  demon_2 :  sh_facet  stocks  {

                int     rules[MAX_STOCKS+1][2];
                int     client_list[MAX_STOCKS+1][MAX_CUSTOMER+1];
                int     number_of_rules;
                int     number_of_client_list;
                int     mailbox[MAX_CUSTOMER+1][MAX_STOCKS+1];
                int     demon_init;


void    init()
{
        int  client_id, monitor_num, stockn, p_buy, p_sell;
        int  client_rule;
        if (demon_init==0) {
                demon_init = 1;
                fprintf(stdout, "\nDAEMON_CITI:        demon initialization: ");
                fprintf(stdout,
"\nDAEMON_CITI:      Enter number of stocks to monitor  ");
                fscanf(stdin, "%d", &monitor_num);
                number_of_rules = monitor_num;
                fprintf(stdout,
"\nDAEMON_CITI:        number of stocks being monitored:  %d",
get_number_of_rules());
                for (int i=1; i<number_of_rules()+1; ++i) {
                        fprintf(stdout,
"\nDAEMON_CITI:        Enter stock number and buying and selling price  ");
                        fflush(stdout);
                        fscanf(stdin, "%d %d %d", &stockn, &p_buy, &p_sell);
                        rules[stockn][BUY] =  p_buy;
                        rules[stockn][SELL] = p_sell;
                        fprintf(stdout,
"\nDAEMON_CITI:        How many clients are associated with this rule?  ");
                        fscanf(stdin, "%d", &client_rule);
                        if (i==1 || (client_rule > number_of_client_list))
                                number_of_client_list = client_rule;
                        for (int j=1; j<client_rule+1; j++){
                                fprintf(stdout,
"\nDAEMON_CITI:      Enter client id: ");
                                fscanf(stdin, "%d", &client_id);
                                client_list[i][j] = client_id;
                          }
                }
        print_out_rules();
        print_out_client_list();
        fprintf(stdout, "initialization of DAEMON_CITI complete\n");
        fflush(stdout);
      }
}

void    run()
{
        int  buy, sell, var_assign;
        for(;;) {
             fprintf(stdout, "\nDAEMON_CITI:        checking stock prices...\n");
             for(int i=1; i<number_of_rules+1; i++) {
                     buy  = rules[i][BUY];
                     sell = rules[i][SELL];
                     if (buy > stock_prices[i]){
                             for(int j=1; j<number_of_client_list()+1;j++)
                                       if (client_list[i][j]!=0)
                                         {
                                              fprintf(stdout,
"\nDAEMON_CITI:        notify client %d to BUY stock %d\n", client_list[i][j], i);
                                              mailbox[client_list[i][j]][i] = BUY;
                                              mailbox[client_list[i][j]][0] = SET;
                                              fprintf(stdout,"%d\n",
```

```
                                                mailbox[client_list[i][j]][i]);
                                              fflush(stdout);
                                              fprintf(stdout,
"\nDAEMON_CITI:        stock %d:  buy price:  %d , cur price: %d\n",
i, buy, stock_prices[i]);
                                              fflush(stdout);
                                              }
                           }

                       if (sell < stock_prices[i]){
                             for(int j=1; j<number_of_client_list+1;j++)
                                if (client_list[i][j]!=0)
                                 {
                                     fprintf(stdout,
"\nDAEMON_CITI:        notify client %d to SELL stock %d\n", client_list[i][j], i);
                                     mailbox[client_list[i][j]][i] = SELL;
                                     mailbox[client_list[i][j]][0] = SET;
                                     fprintf(stdout,"%d\n",
                                             mailbox[client_list[i][j]][i]);
                                     fflush(stdout);
                                     fprintf(stdout,
"\nDAEMON_CITI:        sell price:  %d , cur price: %d\n",
sell, stock_prices[i]);
                                     fflush(stdout);
                                   }
                           }
                   }
           }
      }


     void    print_out_rules()
         {
               for (int i=1; i<number_of_rules+1; ++i)
                 fprintf(stdout,
"\nDAEMON_CITI:        stock: %d  buy: %d  sell: %d\n", i, get_rules[i][BUY], rules[i][SELL]);
               }

     void    print_out_client_list()
         {
                 for (int i=1; i<number_of_rules()+1; ++i){
                           fprintf(stdout,"\nDAEMON_CITI:        stock: %d  ", i);
                 for (int j=1; j<number_of_client_list+1; ++j)
                           fprintf(stdout,
"\nDAEMON_CITI:        %d  ", client_list[i][j])
                           }
                  fprintf(stdout, "\n"); fflush(stdout);
               }
}; end_sh_facet



object  trader_1
{
      facet  trader :  sh_facet stocks, sh_facet demon_1 {

        interface:

            int     trader_number;
            int     portfolio[MAX_STOCKS+1][2];

void
init()
{
        int  i,j, b, s, trad;
        fprintf(stdout, "\nCITICORP:      Enter trader ID   ");
        fscanf(stdin, "%d", &trad);
```

```
        trader_number = trad;
        fprintf(stdout,
"\nCITICORP:      Do you want to want to enter portfolio now   ");
        fscanf(stdin, "%d",&i);
        if (i) {
             fprintf(stdout, "\nCITICORP:      How many stocks in portfolio?   ");
             fscanf(stdin, "%d", &j);
             for (int k=1; k<j+1; k++){
                 fprintf(stdout,
 "\nCITICORP:      for stock %d, enter a buying and selling price\n", k);
                 fscanf(stdin, "%d %d", &b, &s);
                 portfolio[k][BUY] = b;
                 portfolio[k][SELL] = s;}
           }
      }


void
run()
{
        int  j, tmp, t_chk;
        for(;;) {
          fprintf(stdout, "\nCITICORP:      %d: \n", trader_number);
          fflush(stdout);
C_CODE    mon_enter(&m_mailbox_1);
          if ((t_chk=check_stock()) > 0) {
                 for(j=1;j<MAX_STOCKS+1;j++){
                     if ((tmp = check_stock(j))>0) {
                         if (tmp == BUY)  buy_stock(j);
                         else             sell_stock(j);
                         clear_check_bit(j);
                       }
                   }
          clear_check_bit(0);
               }
C_CODE          mon_exit(&m_mailbox_1);
                fprintf(stdout,"%d\n", mailbox[trader_number]][j];
                fflush(stdout);
                if (check_stock() <= 0)
                    fprintf(stdout,"Computing strategies\n");
        }
      }

int
buy_stock(int i)
{
    int       buy_num;
    long int   random();

    buy_num =  ((int)random()) % FLUX;
    if (!buy_num)   ++buy_num;
    fprintf(stdout,
"\nCITICORP:      trader %d has BOUGHT %d shares of stock %d at %d\n",
trader_number, buy_num, i, stock_prices[i]);
    fflush(stdout);
    stock_prices[i] = stock_prices[i]+buy_num;
    return(buy_num);
}

int
sell_stock(int i)
{
    int       sell_num;
    long int   random();
    sell_num =  ((int)random()) % FLUX;
    if (!sell_num)  ++sell_num;
    fprintf(stdout,
```

```
"\nCITICORP:      trader %d has SOLD %d shares of stock %d at %d\n",
trader_number, sell_num, i, stock_prices[i]);
    fflush(stdout);
    stock_prices[i] = stock_prices[i] - sell_num;
    return(sell_num);
  }

int
check_stock()
{
         return(mailbox[trader_number][0]);
}

int
check_stock(int j)
{
         return(mailbox[trader_number][j]);
}

void
clear_check_bit(int i)
{
         mailbox[trader_number][i] = 0;
}
} end_facet
}; end_object


object  trader_2
{
      facet  trader :  sh_facet stocks, sh_facet demon_2 {

        interface:

            int     trader_number;
            int     portfolio[MAX_STOCKS+1][2];

void
init()
{
        int  i,j, b, s, trad;
        fprintf(stdout, "\nCITICORP:      Enter trader ID   ");
        fscanf(stdin, "%d", &trad);
        trader_number = trad;
        fprintf(stdout, "\nCITICORP:      Do you want to want to enter portfolio now   ");
        fscanf(stdin, "%d",&i);
        if (i) {
             fprintf(stdout, "\nCITICORP:      How many stocks in portfolio?   ");
             fscanf(stdin, "%d", &j);
             for (int k=1; k<j+1; k++){
                 fprintf(stdout,
"\nCITICORP:      for stock %d, enter a buying and selling price\n", k);
                 fscanf(stdin, "%d %d", &b, &s);
                 portfolio[k][BUY] = b;
                 portfolio[k][SELL] = s;}
           }
      }


void
run()
{
        int  j, tmp, t_chk;
        for(;;) {
          fprintf(stdout, "\nCITICORP:      %d: \n", trader_number);
          fflush(stdout);
C_CODE    mon_enter(&m_mailbox_1);
          if ((t_chk=check_stock()) > 0) {
                 for(j=1;j<MAX_STOCKS+1;j++){
                     if ((tmp = check_stock(j))>0) {
                         if (tmp == BUY)  buy_stock(j);
                         else             sell_stock(j);
                         clear_check_bit(j);
                       }
                   }
          clear_check_bit(0);
               }
C_CODE          mon_exit(&m_mailbox_1);
                fprintf(stdout,"%d\n", mailbox[trader_number]][j];
                fflush(stdout);
                if (check_stock() <= 0)
                    fprintf(stdout,"Computing strategies\n");
        }
      }

int
buy_stock(int i)
{
    int        buy_num;
    long int   random();

    buy_num =  ((int)random()) % FLUX;
    if (!buy_num)   ++buy_num;
    fprintf(stdout,
 "\nCITICORP:      trader %d has BOUGHT %d shares of stock %d at
%d\n",
trader_number, buy_num, i, stock_prices[i]);
    fflush(stdout);
    stock_prices[i] = stock_prices[i]+buy_num;
    return(buy_num);
}

int
sell_stock(int i)
{
    int        sell_num;
    long int   random();
    sell_num =  ((int)random()) % FLUX;
    if (!sell_num)  ++sell_num;
    fprintf(stdout,
"\nCITICORP:      trader %d has SOLD %d shares of stock %d at %d\n",
trader_number, sell_num, i, stock_prices[i]);
    fflush(stdout);
    stock_prices[i] = stock_prices[i] - sell_num;
    return(sell_num);
  }

int
check_stock()
{
         return(mailbox[trader_number][0]);
}

int
check_stock(int j)
{
         return(mailbox[trader_number][j]);
}

void
clear_check_bit(int i)
{
         mailbox[trader_number][i] = 0;
}
```

```
} end_facet                                                                                                         }
}; end_object

profit_end


C_CODE       sec            e;
C_CODE       demon_1       d1;
C_CODE       trader_1      t1;
C_CODE       demon_2       d2;
C_CODE       trader_2      t2;

profit_main {

       e.init();
       d1.init();
       t1.init();
       d2.init();
       t2.init();

       thread  task_sec;
       thread  task_daem_1;
       thread  task_trad_1;
       thread  task_daem_2;
       thread  task_trad_2;

     }; end_main


void   task_sec()
{
     e.run();
 }


void  task_daem_1()
{
     d1.run();
   }

void   task_trad_1()
{
     t1.run();
 }


void  task_daem_2()
{
     d2.run();
   }

void   task_trad_2()
{
     t2.run();
 }



void begin_day()
{
     fprintf(stdout,
"\n\n\nA DAY OF TRADING on the New York Stock Exchange is about to commence.\007");
     fflush(stdout);
     system("sleep 1");
     fprintf(stdout,"..\007"); fflush(stdout);
     system("sleep 1");
     fprintf(stdout,"..\007"); fflush(stdout);
```