A Software Pattern for Autonomic Object Behavior

John W. Gilbert
Objective Software
1985 W. Henderson Rd., Suite 525
Columbus, Ohio 43220
phone: 614-457-4487
fax: 614-459-7310
email: gilbert@acm.org

July 28, 1996

Abstract

Living beings exhibit behavior in a continous concurrent manner. For example animals and humans have heartbeats and respiration. This is referred to as autonomic behavior. Some machines also operate in this manner. For example process control systems continuously monitor a manufacturing process in real-time and make corrections to the process as required. Software objects often are considered to be inherently autonomous and concurrent, but due the sequential nature of ordinary computers this style of computation is often absent in the software implementation. This paper describes a software pattern for emulating this autonomic behavior which can be profitably used in a variety of systems: maunfacturing, medical monitoring, and safety management.

Keywords
thread, task, concurrent, autonomic, execution, synchronous, asynchronous, access control, state, response, behavior

The description of this pattern is modeled after the Design Patterns book[Gamm94].

Nomenclature

Pattern Name	Classification
PrivateThread	Behavioral

Also Known As
PrivateProcess, MethodProcess, MethodThread, PrivateTask, MethodTask

Intent

Organizing concurrency within application software to be both easy to understand, and easy to implement, is an important aspect of software design. In object-oriented systems, reasonable abstractions can be constructed to address this issue, and help the programmer to identify opportunities for concurrency within an application. The analogy of autonomic human behavior with object behavior is such an abstraction. Using this concept designers can model the objects they find in their problem domain more naturally, thereby, creating an implementation which is more understandable and coincident with the problem structure.

Motivation

All objects exhibit responsive sequential (or reactive) message behavior; that is, they respond to messages sent to them from other objects by executing a method and possibly returning a reply. This is sufficient to model many types of real-world objects, but other, more complex objects like those found in nature and industry, often require that complex and concurrent behavior be modeled. Anthropomorphic objects (animals, humans, etc.) exhibit behavior continuously. For example, the autonomic functions of animals (breathing, heart beat, digestion, etc.) are concurrent with all other behavior. An object modeling an animal would need to exhibit this behavior. Real world entities such as sensors, actuators, and manufacturing processes also exhibit continuous behavior, such as continuous real-time process measurements. This behavior is directly analogous to respiration or heartbeat in an animal. A PrivateThread is a way of providing a mechanism for instantiating this autonomic, or concurrent continuous behavior in software objects.

In the usual object-oriented environments of C++ or Smalltalk, the programming model is single threaded. Work progresses from one method to another, serially. Library extensions to these languages often introduce Process[1] objects. These objects either require a cooperative style of concurrent programming which is difficult to manage and maintain, or the implementation is so heavyweight as to make usage impractical. The PrivateThread pattern avoids re-inventing a system's concurrency mechanisms by encapsulating the underlying operating system services, making concurrency more manifest in the object model, and the work (method execution) appear to be concurrent.

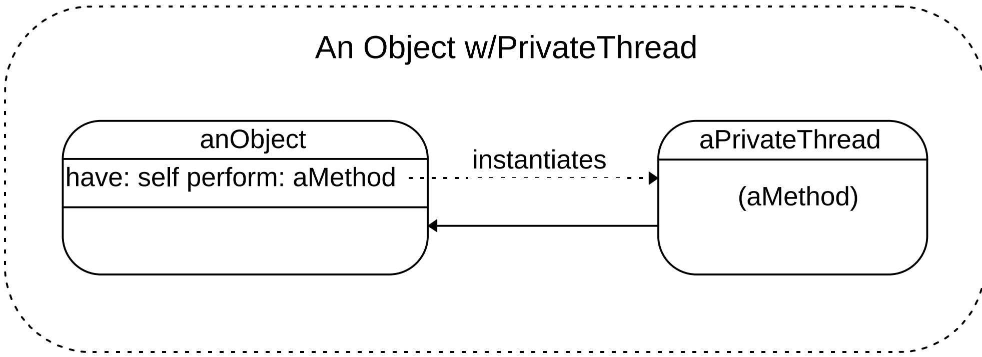
Example

In many industrial processes, sensors are employed to measure properties in a product under manufacture. For example, sensors do one activity all the time: they measure. Sensors must, also, respond to requests for status and perform calibration procedures. A permanent PrivateThread is employed by a sensor's software object to execute the autonomic function of measuring, and at scheduled times, or on request, transitory PrivateThreads are initiated for automated calibration procedures, which collect data over timed intervals to compute averages used in determining calibration constants.

[1] The term *process*, unfortunately, can have a programming construct meaning, as well as a meaning in manufacturing systems. The usage of the programming term has been eliminated except when referring to specific language systems. In those cases the first letter is capitalized.

The Pattern

Structure



In the diagram above, *anObject* encapsulates *aPrivateThread*, hence, the dotted composite structure. The *aPrivateThread* object cannot be affected externally except through *anObject's* public interface. On the other hand, the method executed by *aPrivateThread* can communicate with any collaborators of *anObject*, because, being a method of *anObject*, it has full access to *anObject's* instance variables. This is represented in the diagram by the object reference arrow back to *anObject*.

An object instantiates one or more PrivateThreads, each one to execute one of its methods concurrently; that is, concurrently with the object's reactive behavior, as shown above. A PrivateThread is the encapsulation of an operating system task, or thread resource, combined with a mechanism for delivering messages to other tasks in the system. Since a PrivateThread is private, in the same sense as instance variables, it cannot be accessed except through the parent object's interface, hence, messages cannot be received by the PrivateThread from outside. In effect, the PrivateThread can only do one thing: perform the supplied parent's method on a separate thread of control. Just like any other method, this method can communicate with any other object which is an acquaintance of the parent object. The privacy restriction is only in regard to unsolicited communications from other objects, which is only allowed through the parent object's public interface.

Participants

The PrivateThread class provides the mechanisms and operating system interfaces to support the execution of an object's method by a separate thread. Any object can initiate such an execution by dynamically creating an instance of PrivateThread and assigning it some object's method[2]. The participants in this pattern are its initiator (any object desiring concurrent method execution) and the method being threaded.

[2] This part of PrivateThread is using an internal behavioral pattern similar to the Command pattern in Gamma et al.[Gamm94].

Collaborations

The main collaboration in PrivateThread is with the parent[3] object on whose behalf the separately threaded method is being executed. The extent to which the system provides object abstractions for threads, resources, and concurrency controls, determines the additional objects with which PrivateThread may collaborate. Objects which support some form of subscription and notification also collaborate with PrivateThread in providing results gathering and synchronization with other objects and threads in the system.

Consequences

PrivateThreads provide a useful design abstraction with both positive and negative aspects. PrivateThread does not propose to solve all the problems associated with concurrency. Many issues, such as consistency controls, are intended to be addressed in the implementation. The pattern's intent is to document a concurrency abstration which is commonly used by practictioners.

Positive

- The continuous concurrent behavior of real world entities such as sensors, actuators, manufacturing processes, and anthropomorphic objects can be modeled naturally in software.
- Concurrency is encapsulated within the object which exhibits the continuous behavior.
- Concurrent behavior within an object is protected from outside influences by being controlled through the formal interface of the parent object.
- Multiple, possibly identical, concurrent behaviors (methods) can be maintained by multiple instances of PrivateThread.

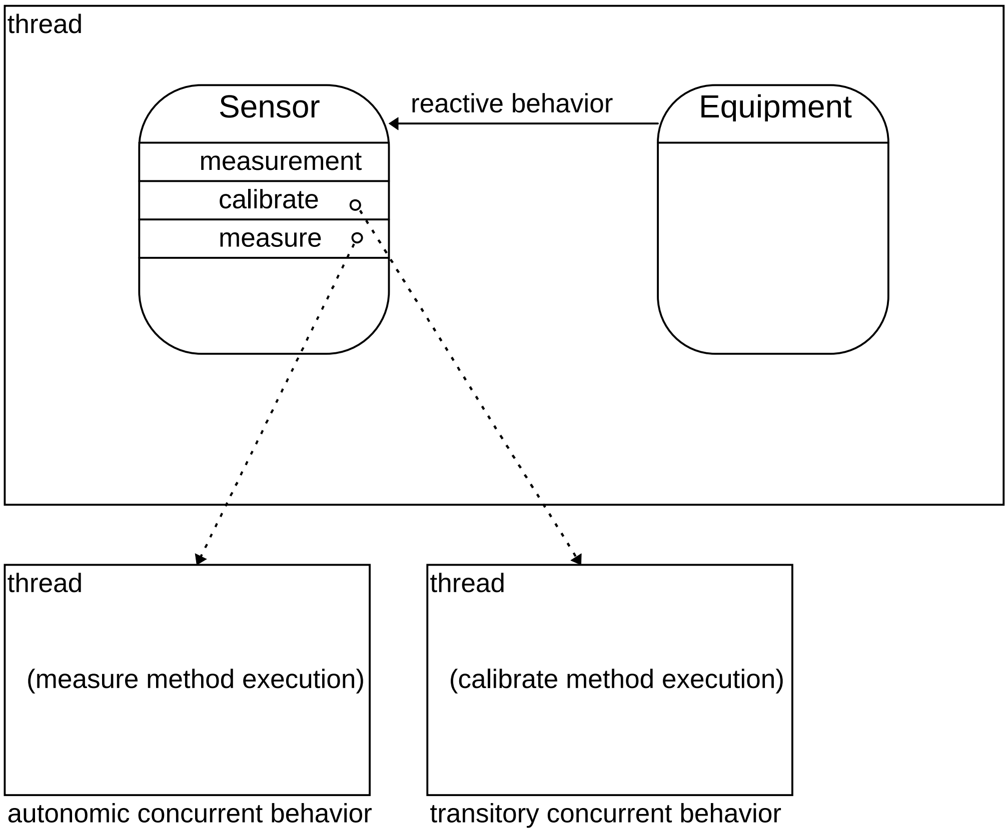
Negative

- PrivateThread does not solve the state consistency problem, but rather, leaves this issue to be resolved during implementation. Work by others, discussed below, have attempted to enforce consistency through either direct language support or pre-compilation processing to automatically add consistency controls where conflicts can be inferred. Therefore, PrivateThread, while designed to be used with any object/method, makes no guarantees about appropriate usage.
- PrivateThread's class implementation is not portable (except in languages which contain a Process model, such as Smalltalk or Java), although its message interface is. The class usually must be implemented to the specifics of the underlying operating system. Users of PrivateThread objects are isolated through this interface from underlying system change, but class writers must adapt to that change.

[3] In this discussion "parent" is used to denote the object which instantiates, owns, and controls the PrivateThread.

- PrivateThread does nothing to prevent the classical deadlock situation. Methods to be run concurrently must be explicitly written to avoid deadlock.

Example



In the example, a sensor is employed to measure some property in a material under manufacture. As shown, there three methods supporting this measurement,

- *measurement* - returns the current value of the measurement.

- *calibrate* - an algorithm which uses current and past measurements to calculate new sensor constants which better reflect the condition of the sensor.

- *measure* - the measurement equation which uses raw input values from the hardware and the calibration constants to produce a time stream of measurement values.

The *Equipment* object can query the *Sensor* object for the latest measurement. Concurrently the *measure* method which is repetitively invoked in its own thread via PrivateThread produces a series of mesurements over time. *Sensor* configuration can be set up to collect batches of these measurements or only retain the the latest. Calibration occurs at regular intervals within the *Sensor*, or upon operator request (not shown) and operates via a separate PrivateThread.

Discussion of Pattern

As stated earlier, an object instantiates one or more PrivateThreads, to execute methods concurrently and is the encapsulation of an operating system thread resource.

PrivateThreads behave as asynchronous activities. They do not return a value as in normal synchronous messaging. Result values, if any, are recorded in the state space (instance variables) of the parent object. Users must access results with proper concurrency controls. This is often realized by using public methods of the parent object which have been written with the necessary access controls. Alternatively, results can be delivered by direct transfer to a requestor via a delegated message supplied by the requestor to the parent object. This kind of behavior is supported by a Subscription/Notification pattern such as that suggested by Wilhelm[Wilh93], and somewhat similar to the combination of the Observer and Command patterns of Gamma et al.[Gamm94].

PrivateThreads can be synchronized with other activities through a signalling interface. For example, a user could ask to be sent one of its own messages on the event of PrivateThread's method completion, or at its initiation, or on any other prearranged event. This, as with the results transfer description above, relies on a Subscription/Notification pattern described in the references, previously indicated.

Any method of an object can be run in a PrivateThread if access to the object's instance variables are done in a concurrently safe manner. Usually, however, particular methods are intended to be run in PrivateThreads and are carefully coded to be safe, and to take advantage of concurrency. Partitioning the state of the parent object to isolate concurrent state usage can help in reducing the burden of access controls. Only the state which is relevant need be locked.

Implementation

Requirements

For this pattern to be practical, the execution environment must provide a fairly lightweight threading environment and some form of access controls, such as lock-flags or semaphores. Any method can be used within a PrivateThread, but in order to be state safe, methods which change object state must be written using access controls. This pattern also assumes that the underlying code libraries are reentrant, so that unforeseen usage of library objects are thread safe, and support concurrent data access.

A PrivateThread maintains at least two instance variables: one for the message associated with the method delegated to it, and one for the receiver of that message. Other instance variables, depending on implementation decisions, can be added to control execution period and status, provide storage for result values and notifications, and log execution statistics for debugging purposes.

Sample Code

Various methods can be implemented to manage PrivateThreads. The actual code for these methods is highly dependent on the underlying system interfaces, e.g. operating system services, API, etc. Below is a list of the types of methods one would expect the PrivateThread class to implement. (The samples are in Objective-C and read very much like Smalltalk. The brackets indicate the beginning of an object message expression, colons indicate method arguments, and the assignment syntax is C.)

Creation

Creates an instance prepared to execute the supplied method

```
aPrivateThread = [PrivateThread have:self perform:"calibrate"];
```

Execution Policy Specification

Sets up cyclic execution

```
aPrivateThread = [PrivateThread have: self perform:"calibrate" every: .5 :SECONDS];
```

Executes only once

```
aPrivateThread = [PrivateThread have: self performOnce:"calibrate" ];
```

Execution Control

```
[aPrivateThread stop];                  // arrests execution
[aPrivateThread kill];                  // destroys task
[aPrivateThread syncTo: aTime];         // time synchronizes execution
[aPrivateThread runEvery: 220 :MSEC];   // resyncs execution to cyclic period
```

External Synchronization and Results Exchange

Notifies a user of execution completion using user supplied method

```
[aPrivateThread signal: aConcernedObject :ON_COMPLETION withMessage:"start"];
```

Request for delivery of concurrent method's result to a user using a user supplied method.

```
[aPrivateThread deliverResultTo: aUser withMessage:"testResult:"];
```

Typical Usage Example

To Create a new sensor and initialize it

```
aSensor = [Sensor new];
[aSensor initialize];
```

Within 'initialize' method of Sensor

```
aMeasurementThread = [PrivateThread have: self perform: "measure"];
[aMeasurementThread runEvery: 100 :MSEC];
// The PrivateThread object will now run the method "measure" every 100 milliseconds
// on behalf of aSensor.
```

To halt the measurement task, use the following

```
[aMeasurementThread stop];
```

And to get rid of the PrivateThread object

```
[aMeasurementThread kill];
aMeasurementThread = nil;
```

Discussion

A PrivateThread can be instantiated as a one-shot, delayed, or cyclic task, but the execution behavior of the PrivateThread is dependent upon the behavior of its executed method. Some methods executed as PrivateThreads may include interactions with the system's timing services, or initiate notification requests, which can influence or delay the thread's expected timing characteristics. The more tightly integrated the implementation of PrivateThread is with the underlying system, the easier it is to control these temporal characteristics, or to detect misaligned events and other timing concerns, but at the expense of portability.

The methods executed by PrivateThreads may choose to update their parent's object state, directly, but only via appropriate concurrency controls, requiring the state to be locked during usage. Alternately, and ensuring safety, a synchronous message may be sent from the method utilizing formal access/update methods in the parent object to change state. Consequently, in order to write code which is ignorant of concurrency, all state changes (to instance variables) must be done via these access methods. Otherwise, the parent object's method executed by a PrivateThread must be controlled via a semaphore, or other locking mechanism, for the entire length of its execution. This would turn the parent object into a MonitorObject, effectively an encapsulation of Hoare's monitor[Hoar74]. The use of synchronous messages creates the possibility of deadlock between two threads. Although a PrivateThread is encapsulated by an object, it operates on that object's state. This puts it in potential conflict with other concurrent usages of the object, resulting in potential deadlock situations. Care must be taken in writing object methods, which run concurrently, to avoid deadlock.

Usage and Related Work

Usage

Use this pattern when modeling real world objects that exhibit concurrent behaviors. Objects which implement parallel computations can also use this pattern to simulate parallelism. Many types of monitoring applications can use PrivateThread.

This pattern has been used extensively in a real-time industrial measurement and control system since 1986[Gilb93]. That system uses PrivateThreads to accomplish sensor measurements, safety monitoring, timed response, and periodic online calibration procedures.

Related Patterns

Patterns relating to Exceptions and Error Notifications can be used in controlling and managing PrivateThread execution. Any pattern which encapsulates a system's thread of control mechanism would serve as a basis for PrivateThread. Patterns used to supply requested results on event changes, such as Subscription/Notification or Observer/Command mentioned earlier, can be used to deliver method results from within a PrivateThread. A pattern like Command can be used to encapsulate the message and receiver used by PrivateThread in executing the method delegated to it.

Related Work and Discussion

All of the systems discussed below allow objects to have multiple threads of control private to themselves, and many attempt to deal with consistency issues in various ways incremental to PrivateThread.

The concept of concurrency within objects is mentioned in Yonezawa and Tokoro[Yone87][4], but not in any detail. Caromel[Caro93] has done much work in the area of concurrent objects, incorporating some of the features of PrivateThreads, but his scheme requires objects to inherit from a common Process class, as well as an application class, requiring multiple inheritance, and concurrent methods must be pre-declared[Caro89]. PrivateThread makes no assumptions about inheritance and is very dynamic in its usage of concurrency, i.e., any object can be given concurrent behavior any time by “assigning” any one of its methods to a thread (assuming the implementation language permits such assignment). This obviously carries the responsibility of only initiating concurrently appropriate methods, ones which are state safe, and therefore carries the risk of runtime type conflicts, or object consistency problems (the usual trade-off between static and dynamic programming).

This problem of object consistency in concurrent systems is often tackled at the language level. The MELD[Kais92] language which gives support for multiple threads of control within an object uses Atomic Blocks (an encapsulation of critical sections) to enforce object state consistency. The responsibility for the identification and creation of Atomic Blocks is with the programmer. The Ceiffel language[Löhr92] goes a little further by introducing the idea of Concurrency Annotations which cue a special compiler to generate correct concurrency controls. In that system, these same methods can be used sequentially, although only through re-compilation. Again, as with PrivateThreads, the programmer must pre-plan consistency protection.

To eliminate the burden of programmers having to manage and pre-plan concurrent access and object consistency, compilers could generate code to automatically lock individual instance variables, or lock whole objects. The latter is a MonitorObject and serializes execution through the object. This effectively eliminates internal object concurrency, but leads to the ideas supporting Active Objects[Nier87][Agha90][Kafu91], which use a different concurrency model to implement a high degree of concurrent behavior[5]. In practice, systems sometimes use both models[Gilb93][Øste88].

---

[4] This work, while a bit dated, is a good beginning text on the subject of object-oriented concurrency. A later volume[Agha93] brings the subject up to date.

[5] The Active Object model has recently been described in pattern format by Lavender and Schmidt[Lave95].

Acknowledgements

Suggestions and Review
Tom Love, WorldStreet Corp.
Bob Wilhelm, Objective Control
Ralph Kolewe, Eridani Inc.
Bran Selic, ObjecTime Ltd.
Participants of the CPD Workshop, OOPSLA'95

References

[Agha90] Agha, G. "Concurrent Object-Oriented Programming," *CACM*, September 1990.

[Agha93] Agha, G., Wegner, P., and Yonezawa, A. Research Directions in Object-Oriented Concurrent Programming, MIT Press 1993.

[Caro89] Caromel, D. "Service, Asynchrony, and Wait-By-Necessity," *Journal of Object-Oriented Programming*, Nov/Dec 1989

[Caro93] Caromel, D. "Toward a Method of Object-Oriented Concurrent Programming," *CACM*, September 1993.

[Gamm94] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Object-Oriented Software Architecture. Addison-Wesley, 1994.

[Gilb93] Gilbert, J., Wilhelm, R. "A Concurrent Object Model for an Industrial Process-Control Application," *Journal of Object-Oriented Programming*, Nov/Dec 1993.

[Hoar74] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept," *CACM*, Oct. 1974.

[Kafu91] Kafura, D. "Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Languages," *ACM OOPS Messenger*, April 1991.

[Kais92] Kaiser, G., Hseush, W., Popovich, S., and Wu, S. "Multiple Concurrency Control Policies in an Object-Oriented Programming System," Columbia University, CUCS-020-90, revised January 1992.

[Lave95] Lavender, G. and Schmidt, D. "Active Object: an Object Behavioral Pattern for Concurrent Programming," *Proceedings Pattern Languages of Programming*, September 1995.

[Löhr92] Löhr, K-P. "Concurrency Annotations," *Proceedings OOPSLA'92, ACM SIGPLAN Notices*, October 1992.

[Nier87] Nierstrasz, O. “Active Objects in Hybrid,” *Proceedings OOPSLA’87, ACM SIGPLAN Notices*, December 1987.

[Øste88] Østerbye, K. “Active Objects: An Access Oriented Framework for Object-Oriented Languages,” *Journal of Object-Oriented Programming*, June/July 1988.

[Wilh93] Wilhelm, R., “Publish and Subscribe with User Specified Action,” Patterns Workshop, OOPSLA’93.

[Yone87] Yonezawa, A. and Tokoro, M. <u>Object-Oriented Concurrent Programming</u>, Introduction, p. 3, MIT Press, 1987.