Real-Time Design Patterns

Bruce Powel Douglass, Ph.D.
Chief Methodology Scientist
I-Logix

Some of the contents of this paper are adapted from the author's book *Real-Time UML: Developing Efficient Objects for Embedded Systems* published by Addison-Wesley, 1998.

Introduction

Many of us have had the experience of working with a truly great software designer. They have the ability to look at a really hard problem and seemingly with no effort construct an elegant and practical solution. Afterwards we all slap our foreheads and say "Of Course! It's so obvious!" How do those Great Designers actually come up with these great designs in the first place?

By training, I am a neurophysiologist. During my medical school years, I primarily studied information processing in biological neural systems[1]. The process of coming up with Good Designs proceeds in 3 phases:

- Internalization
- Pattern Matching
- Sequential Analysis

The first phase, *internalization*, is a linear process of gathering information. What are the functions of the design? What are its constraints? What aspects of the design should be optimized at the expense of others? This is a process of gathering and organizing information about the system under design.

The second phase, *pattern matching*, is an inherently nonlinear process and is performed almost exclusively by the subconscious mind. It is exactly this kind of thinking that has taken place when The Answer suddenly comes to you in the shower or when you wake in the middle of the night. It often happens to me when I go for a run. I even go so far as to classify problems by how far I think I'll have to run to come up with a solution. I might tell my boss "Gee, that's a 10-mile problem. I'll see you in a couple of hours – Bye![2]"

I believe that subconsciously our rather impressive pattern matching apparatus goes to work on a pre-analyzed problem (that's where the internalization phase comes in) and does "best fit" pattern matching comparing thousands of patterns while our conscious

[1] Mostly in the subesophageal ganglion of the *Hirudo medicinalis* (a.k.a., the medicinal leech), which, behaviorally has more in common with marketing folks than engineers, but I believe the same rules apply.
[2] You can't believe how hard it is to sell this work flow concept to most managers!

mind is off doing other things. Only when it finds a close-enough match (a potential design solution) does it signal the conscious mind.

Just because the subconscious thinks it has found a good solution is no guarantee as to the quality of the proposed solution. Once identified, it is up to the linear processing system of the brain to take the proposed solution and see if it does in fact meet our criteria. This is the process of *sequential analysis* that is applied to a proposed solution. Most often, this takes the form of mentally applying scenarios against the design pattern to ensure that it in fact meets the necessary criteria and optimizes all the right things.

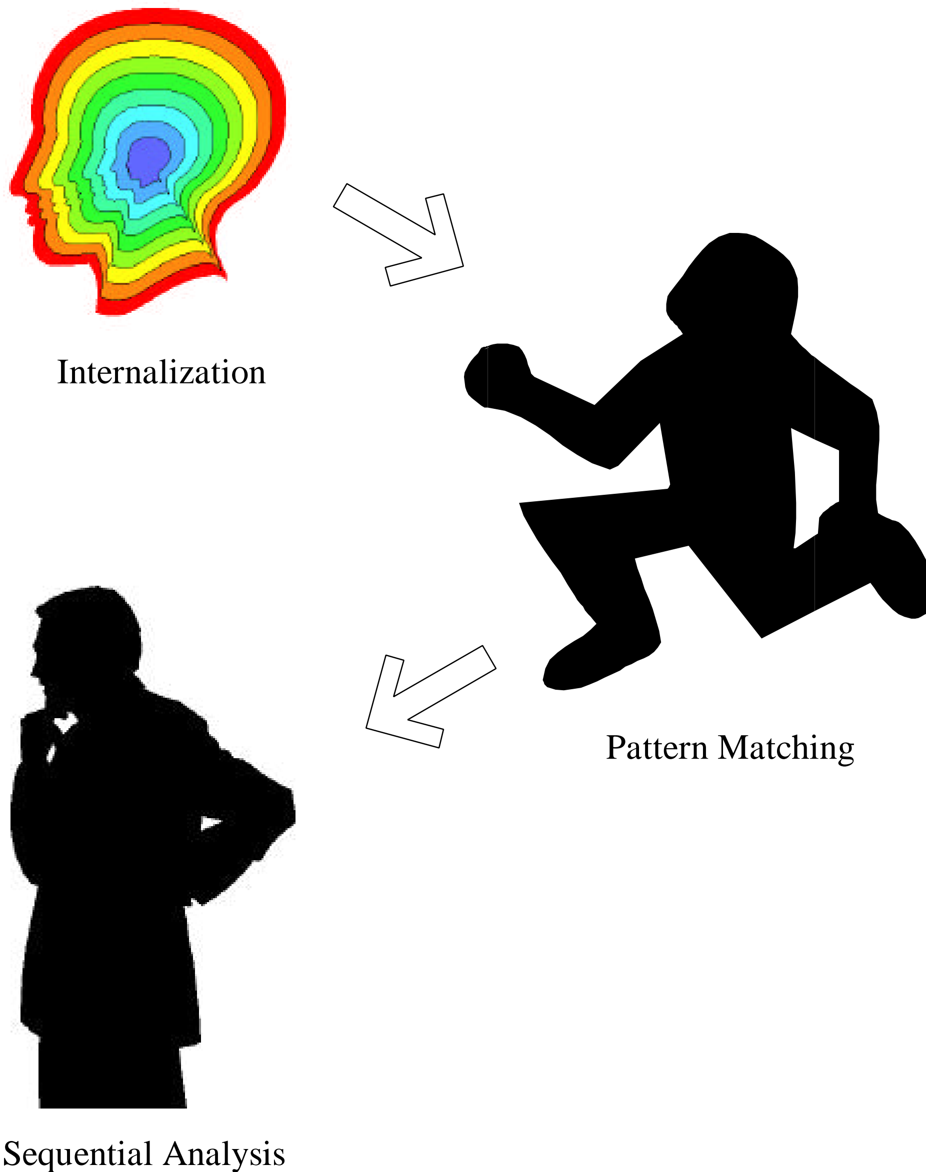


Figure 1: The Creative Design Process

It's the part in the middle, the *pattern matching* phase where a "miracle occurs." However, I don't believe this to be a miracle. I believe that the Great Designers have internalized a vast store of design patterns through which they can subconsciously sift.

What is a Design Pattern?

Design patterns have been the target of a great deal of research in the last few years. A design pattern is a general solution to a commonly occurring problem. They are composed of three parts: a problem context, a generalized approach to a solution, and a set of consequences.

Design patterns are usually constructed by extracting out the things in common from a large set of specific instances. This process is called *abstraction* or *inductive reasoning*. For example, C++ programs are riddled[3] with pointers. There are a number of commonly occurring problems with pointers:

- If pointers go out of scope without the memory to which they point being recovered, the memory is no longer accessible (memory leak)
- Even if programmers remember to match up every *new* with a corresponding *delete*, if an exception is thrown before reaching the delete, a memory leak will occur
- If pointers are used prior to being initialized, they can corrupt system memory (uninitialized pointers)
- If pointers are used after the memory to which they point has been recovered, they can corrupt memory that has been reused by the system (dangling pointer)
- If multiple pointers to the same memory location are released, this can corrupt the heap beyond repair (multiple delete)

So here we have a problem context: pointers are enormously useful for all kinds of programming tasks, but their use can lead to errors because there are no safeguards built into the language.

A generalized approach to a solution is to wrap raw pointers used within functions inside of a class. Let's see how the use of such smart pointer classes helps:

- Memory leak: Raw pointers do not have destructors that guarantee proper memory release. However, when objects go out of scope, their destructors are called. This allows memory to be properly released regardless of how the object goes out of scope.
- Uninitialized pointer: a class can use attributes and methods to ensure that its preconditional invariants are met prior to use. Specifically, a smart pointer object can check to see whether or not it has been initialized.
- Dangling pointer: Once the memory pointed to has been released, the smart pointer can remember that fact and disallow further access.

[3] I mean that in every sense of the word.

- Multiple delete: If several pointers to the same storage are used, a smart pointer can keep track of how many pointers there are and automatically delete the referenced memory if and only if the last pointer is deleted.

The last characteristic of a design pattern is the set of consequences of the pattern. Specifically

- there is a small amount of overhead associated with the smart pointer:
  - Prior to each access the pointer should be checked to ensure that it is currently valid
  - A small amount of memory must be used to keep a reference count to track the number of currently valid pointers to the referenced memory
- Discipline must be enforced so that no one inadvertently uses raw pointers to the same memory.

Phases of Design

I break up design into three phases: architectural, mechanistic, and detailed design. Architectural design is concerned with large-scale strategic decisions that have broad and widely-ranging effects. Architectural decisions include the placement of software modules on different processors, real-time scheduling policies, identification of concurrency models, and inter-processor and inter-thread communication.

Mechanistic design is so named because it deals with the construction of mechanisms – groups of objects that collaborate together for medium-sized purposes. Typically, mechanisms include a small number of objects, from 2 to a dozen or so.

Detailed design is concerned with the details of the object innards – data structuring and typing, internal algorithms, visibility and interfaces.

Notation

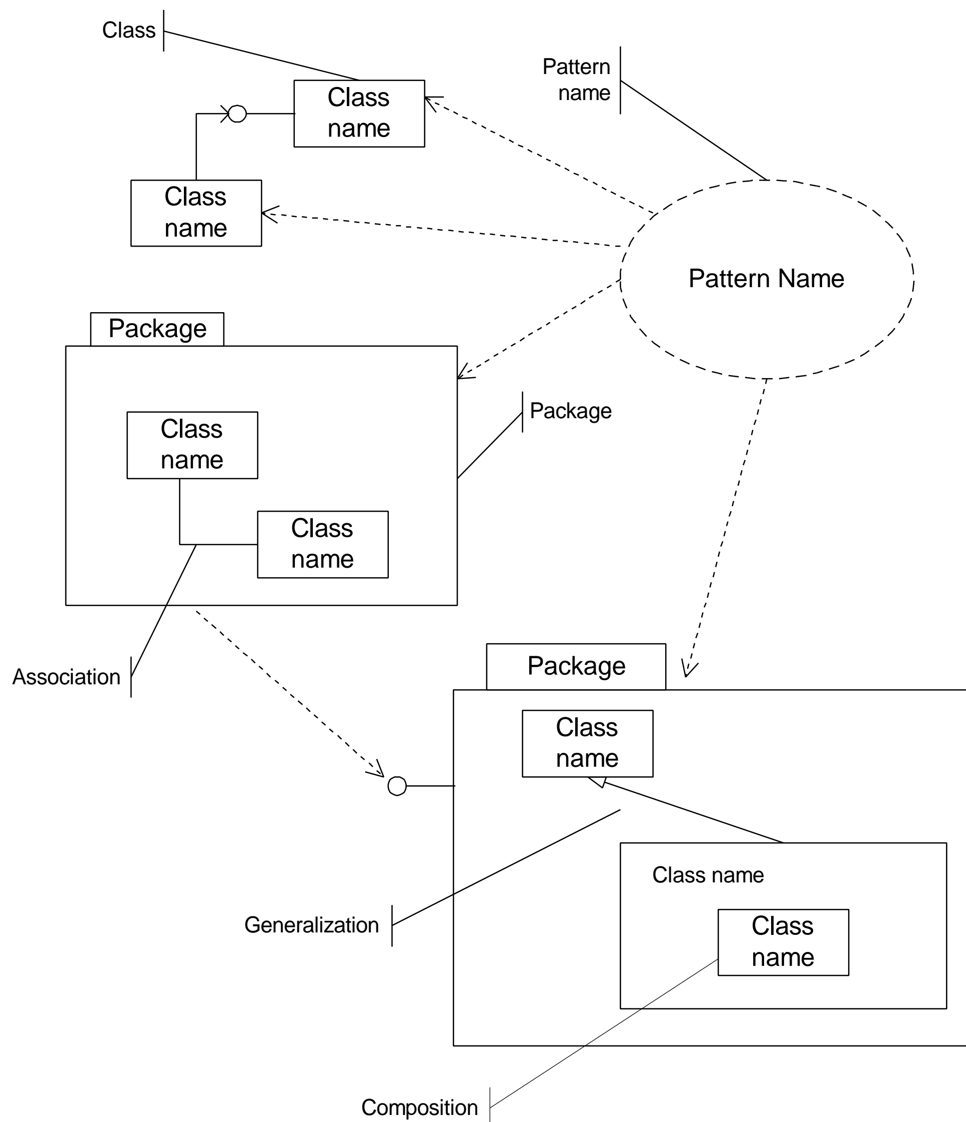
The UML provides a notation for design patterns but this notation is targeted primarily towards mechanistic design patterns. The reason for this is that this is the area receiving the lion’s share of research interest. Nevertheless, the notational schema works for architectural design as well. In this paper, we will not discuss detailed design patterns in order to limit the size of this paper.

The UML pattern notation is based on the UML class diagram. The elements of most interest are:

- Pattern (shown as a named oval)
- Class (shown as a named rectangle)
- Object (shown as a named rectangle in which the name is underlined)

- Package (shown as a tabbed folder)
- Relation
  - Association (shown as a line connecting classes or objects)
  - Aggregation (“has-a” – shown as an association with a diamond at the owner-end)
  - Composition (a strong form of aggregation – shown with containment of classes within classes or with filled diamonds)
  - Generalization (shown as an open arrow pointing to the more general class)
  - Refinement (e.g. instantiation of templates into classes)

A sample design pattern looks like figure 1.



UML Design Pattern Notation

Figure 2: UML Design Pattern Notation

We won't go into the details of the UML notation or semantics (for more detail, see [1] or the other references). Since the UML uses a class diagram to show design patterns, any of the advanced UML features that can appear on a class diagram, such as refinement and dependency relations, stereotypes, notes, etc., can also appear on a design pattern diagram.

Architectural Design Patterns

Architectural design is the identification and definition of large-scale design strategies. These strategic decisions determine how major architectural pieces of the system will be structured, mapped to physical devices, and interact with each other. Such the effect of such design decisions is widespread and affects most or all components. Some of these patterns may be obvious while others seem rather opaque[4].

We'll discuss the following architectural design patterns:

Category	Pattern Name	Purpose
Execution Control	Preemptive Multitasking	Control the execution of tasks based on their priority and readiness to run (single processor).
	Cyclic Executive	Control the execution of tasks based on the proscribed sequence of activities.
	Time Slicing	Control of the execution of tasks based on a fairness doctrine.
	Cooperative Multitasking	Control of the execution of tasks based on mutual agreement among tasks.
Communications	Master-Slave	Multi-processor communication driven by a single master processor querying slave processor nodes.
	Time-Division Multiplexing Access	Multi-processor communication driven by passing a logical time token among multiple processors on a single bus.
	Bus-mastered	Multi-processor communication in which nodes arbitrate for control of the bus.
Reuse	Microkernel	Structuring of a system into a set of layers
Distributed Systems	Proxy	Isolation of a subject away from its client when they are in different address spaces
	Broker	More general form of Proxy used when the addresses of subjects are not known at compile-time.

[4] One of *Douglass' Law of the Universe* (Volume 1) reads "One man's 'Of Course!' is another man's 'Huh?'."

	Asymmetric Multiprocessing	Processor nodes have dedicated processing assignments.
	Symmetric Multiprocessing	Task assignment of processor nodes dynamically allocated based on current processor loading
	Semi-symmetric Multiprocessing	Task assignment of processor nodes allocated at boot time based on available processor capability
Resource	Static Allocation	Pre-allocation of all objects to avoid time overhead for heap management and memory fragmentation
	Fixed Size Allocation	Allocation of fixed sized blocks from one of potentially several different heaps to avoid memory fragmentation
	Priority Ceiling	Used with priority-based preemptive multitasking to avoid unbounded priority inversion
Safety & Reliability	Homogeneous Redundancy	System with multiple identical processing channels for protection against random faults
	Heterogeneous Redundancy	System with multiple diverse processing channels for protection against random and systematic faults
	Sanity Check	Heterogeneous pattern in which one channel performs the primary processing and another performs a lighter-weight reasonableness check on the primary
	Monitor-Actuator	Heterogeneous pattern in which one channel performs actuation while the other monitors the performance of the actuator channel
	Watchdog	Pattern in which a centralized watchdog must be stroked on a regular basis or corrective action will be taken
	Safety Executive	Pattern in which a centralized safety monitor coordinates identification and recovery from faults.

Other patterns are, of course, available. The interested reader is encouraged to examine references 1, 2, 7-11 or look on the Patterns Home Page at http://st-www.cs.uiuc.edu/users/patterns/patterns.html.

Execution Control Patterns

Execution control patterns deal with the policy by which tasks are executed in a multitasking system. This is normally executed by the RTOS, if present. Most RTOSes

offer a variety of scheduling options. The most important of these are listed here as execution control patterns.

The graphical representation of all the execution control patterns looks essentially the same, as shown in Figure 3.

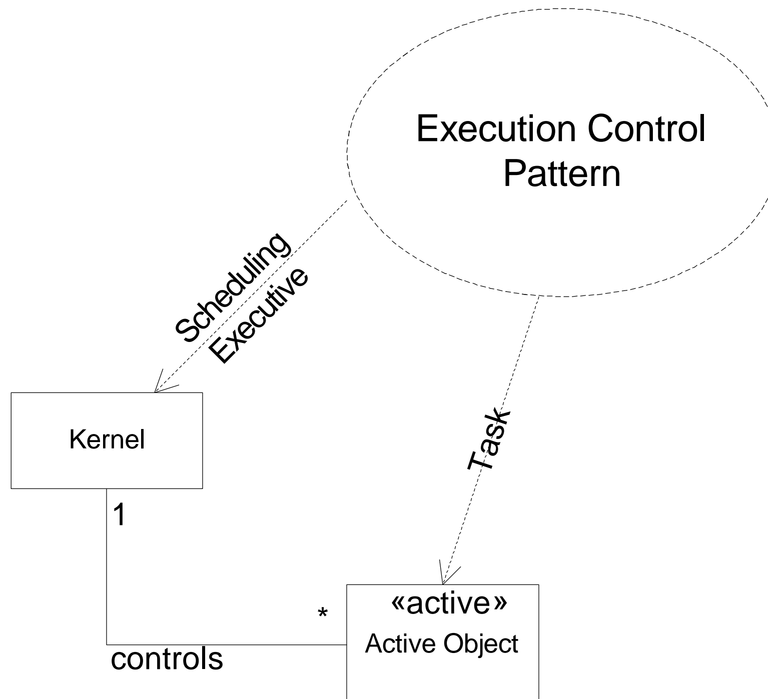


Figure 3: Execution Control Pattern

The primary difference occurs in the policy used for the selection of the currently executing task.

Preemptive Multitasking Pattern

This pattern uses a task's priority and readiness to run as the determining factors controlling when it will be run. Priority-based scheduling systems operate it one of three primary modes: static priority systems, semi-static or dynamic priority systems.

In a static system, a task's priority is determined at compile time and is not changed during execution. This has the advantages of simplicity of implementation and simplicity of analysis. The most common way of selecting task priority is based on the period of the task, or, for asynchronous event-driven tasks, the minimum arrival time between initiating events. This is called *rate monotonic scheduling* (RMS). Static scheduling systems may be analyzed for schedulability using mathematical techniques such as *rate monotonic analysis* (see [2] and [12] for more information).

Semi-static priority systems assign a task a nominal priority but adjust the priority based on the desire to limit priority inversion. This is the essence of the priority ceiling pattern, shown later in this paper.

Dynamic priority systems assign task priority at run-time based on one of several possible strategies. The two most common dynamic priority strategies are

- Earliest Deadline
- Least Laxity
- Maximum Urgency First

In Earliest Deadline (ED) scheduling, tasks are selected for execution based on which has the closest deadline. This algorithm is said to be *dynamic* because task scheduling can not be determined at design time, but only when the system runs. In this algorithm, a set of tasks is schedulable if the sum of the task loadings is less than 100%. This algorithm is optimal in the sense that if it is schedulable by other algorithms, then it is also schedulable by ED. However, ED is not stable; if the total task load rises above 100%, then at least one task will miss its deadline, and it is not possible in general to predict which task will fail. This algorithm requires additional run-time overhead because the scheduler must check all waiting tasks for their next deadline on a frequent basis. In addition, there are no formal methods to prove schedulability before the system is implemented.

Laxity for a task is defined to be the time to deadline minus the task execution time remaining. Clearly, a task with a negative laxity has already missed its deadline. The algorithm schedules tasks in ascending order of their laxity. The difficulty is that during run-time, the system must know expected execution time and also track total time a task has been executing in order to compute its laxity. While this is not conceptually difficult, it means that designers and implementers must identify the deadlines and execution times for the tasks and update the information for the scheduler every time they modify the system. In a system with hard and soft deadlines, the Least Laxity (LL) algorithm must be merged with another so that hard deadlines can be met at the expense of tasks that must meet average response time requirements (see MUF, below). LL has the same disadvantages as the ED algorithm: it is not stable, it adds run-time overhead over what is required for static scheduling, and schedulability of tasks cannot be proven formally.

Maximum Urgency First (MUF) scheduling is a hybrid of RMS and LL. Tasks are initially ordered by period, as in RMS. An additional binary task parameter, critcality, is added. The first n tasks of high criticality that load under 100% become the critical task set. It is this set to which the Least Laxity Scheduling is applied. Only if no critical tasks are waiting to run are tasks from the noncritical task set scheduled. Because MU has a critical set based on RMS, it can be structured so that no critical tasks will fail to meet their deadlines.

Cyclic Executive Pattern

In the cyclic executive pattern the kernel (commonly called the *executive* in this case) executes the tasks in a prescribed sequence. Cyclic executives have the advantage that they are “brain-dead” simple to implement and are particularly effective for simple repetitive tasking problems. However, they are not efficient for systems that must react to asynchronous events and not optimal in their use of time. There have been well-publicized cases of systems that could not be scheduled with a cyclic executive but were successfully scheduled using preemptive scheduling. Another disadvantage of the cyclic executive pattern is that any change in the executive time of any task usually requires a substantial tuning effort to optimize the timeliness of responses. Further if the system slips its schedule, there is no guarantee or control over which task will miss its deadline preferentially.

Time Slicing Pattern

The kernel in the time slicing pattern executes each task in a round-robin fashion, giving each task a specific period of time in which to run. When the task’s time budget for the cycle is exhausted, the task is preempted and the next task in the queue is started. Time slicing the same advantages of the cyclic executive but is more time based. Thus it becomes simpler to ensure that periodic tasks are handled in a timely fashion. However, this pattern also suffers from similar problems as the cyclic executive. Additionally, the time slicing pattern doesn’t “scale up” to large numbers of tasks well because the slice for each task becomes proportionally smaller as tasks are added.

Cooperative Multitasking Pattern

The cooperative multitasking pattern relies on tasks to voluntarily relinquish control in order to allow other tasks to run. One advantage to this pattern is the ability for tasks to control their own destiny once they are started. Specifically, they can implement a “run-to-completion” policy. A significant downside for this pattern is the ability of a single badly-behaved task to halt all task execution in the entire system. Additionally, timing analysis of cooperatively multitasking systems is difficult.

Communications Control Patterns

In systems distributed across more than a single processor, the various processor nodes must communicate amongst themselves to achieve system goals. Communication control patterns provide different solutions to how the control of this communication should be carried out.

Master-Slave Pattern

The Master-Slave communications pattern is used when a single node (the *master*) initiates all communications. The other nodes (*slaves*) respond to queries only when asked. The advantage of this communication control pattern is that it is simple to implement. Specifically, bus arbitration issues don’t arise because the master has total

control. This means that the arbitration overhead (which can be up to 75% of the total bandwidth) is reduced to zero, making the most bandwidth available for actual messaging. The downside of this pattern is that it doesn't scale up to large number of nodes well and performance may be inadequate for asynchronous event handling.

Time-Division Multiplexing Access Pattern

In the TDMA pattern, a logical "Ok-to-Speak" token is passed from node to node based on time. This requires that a stable clock time-base and is sensitive to time drift among nodes. This pattern is simple and efficient because little protocol overhead is required. On the other hand, like the Master-Slave communication pattern, the TDMA may not provide timely response to asynchronous events and doesn't scale up to large numbers of nodes. The TDMA pattern is used in extensively in satellite communications.

Bus-Mastered Pattern

There are many different categories of bus mastered protocol patterns. For example, Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is commonly used in computer networks. Any node can begin speaking as long as the bus is currently unused. In the event of a collision (more than one node begins speaking at the same time), both nodes back off and retry at some random time later. Bit dominance protocols pre-declare a "winner" in the event of a collision. The Control Area Network is an example of a bit dominance bus mastered pattern protocol.

Reuse Patterns

Microkernel Pattern

Only a single reuse pattern is provided here, but many are available. The Microkernel pattern structures the software architecture into a set of layers. The layers are arranged as a set of client-server associations in which the upper layers may call services of the lower layers but not vice versa.

This organization has a number of advantages.

- Portability (reuse on different hardware platforms) is enhanced because the hardware details are isolated in the lowest layers.
- Reuse in different applications is enhanced because the upper layers isolate the application away from the lower layers.
- Scaled-down applications are possible by simply excluding some of the layers.

A closed-layered architecture means that each layer can only call the services of the layer immediately below it. In an open-layered architecture, a layer can call services in any layer beneath it.

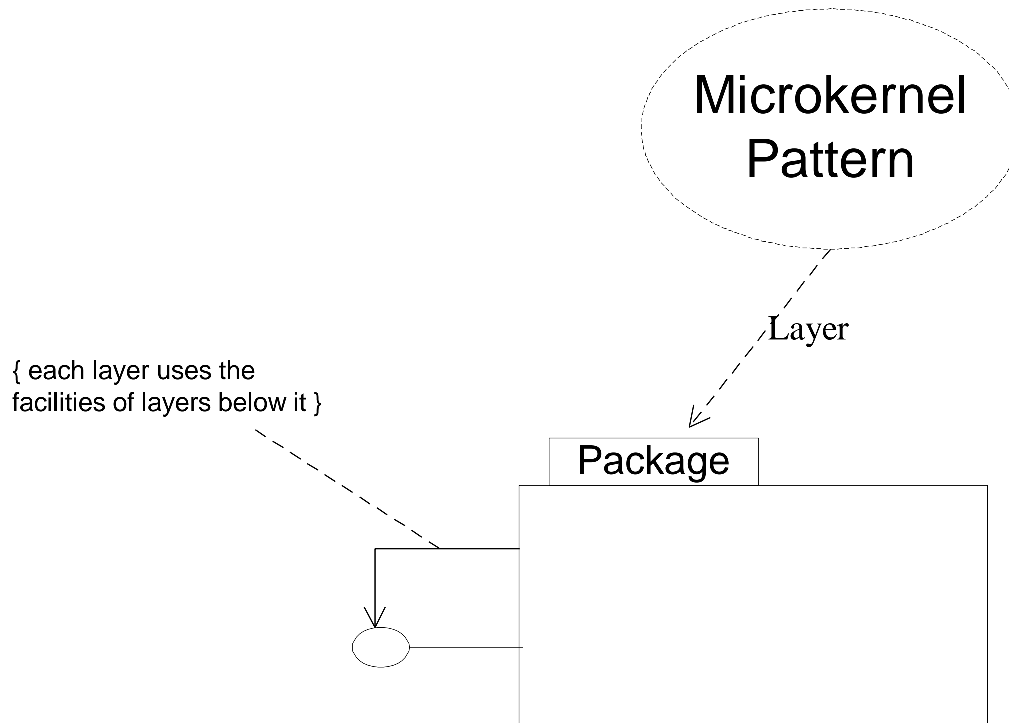


Figure 4: Microkernel Pattern

Distributed Systems Patterns

In this context, a distributed system is one which multiple processors that must collaborate closely together to achieve broad system goals. A number of patterns are useful in such distributed systems.

Proxy Pattern

In many embedded systems data from a single sensor is used by multiple clients who reside in a different address space (task space or processor). The naïve approach to this problem is to have each client capable of tracking down and requesting the data from the data server. This is problematic because if the characteristics of the remote server change, each client must be updated as well.

The Proxy pattern solves this pattern by using a local stand-in for the remote data server, called a proxy. The proxy encapsulates the information necessary to contact the real data server and get up-to-date data. Meanwhile the local clients can directly call the proxy to get the data but they remain decoupled from the remote data server. The client may link to the proxy either by calling it when they need the data, or through the implementation of callbacks. If a callback strategy is used, then the proxy uses one of several strategies to determine when the client should be updated. The most common strategies are

- periodic
- episodic
- epi-periodic

Periodic update strategy just updates the clients every so often, whether or not the data has changed. An episodic update policy notifies the client only when the data changes. The epi-periodic policy notifies the clients using a combination of both.

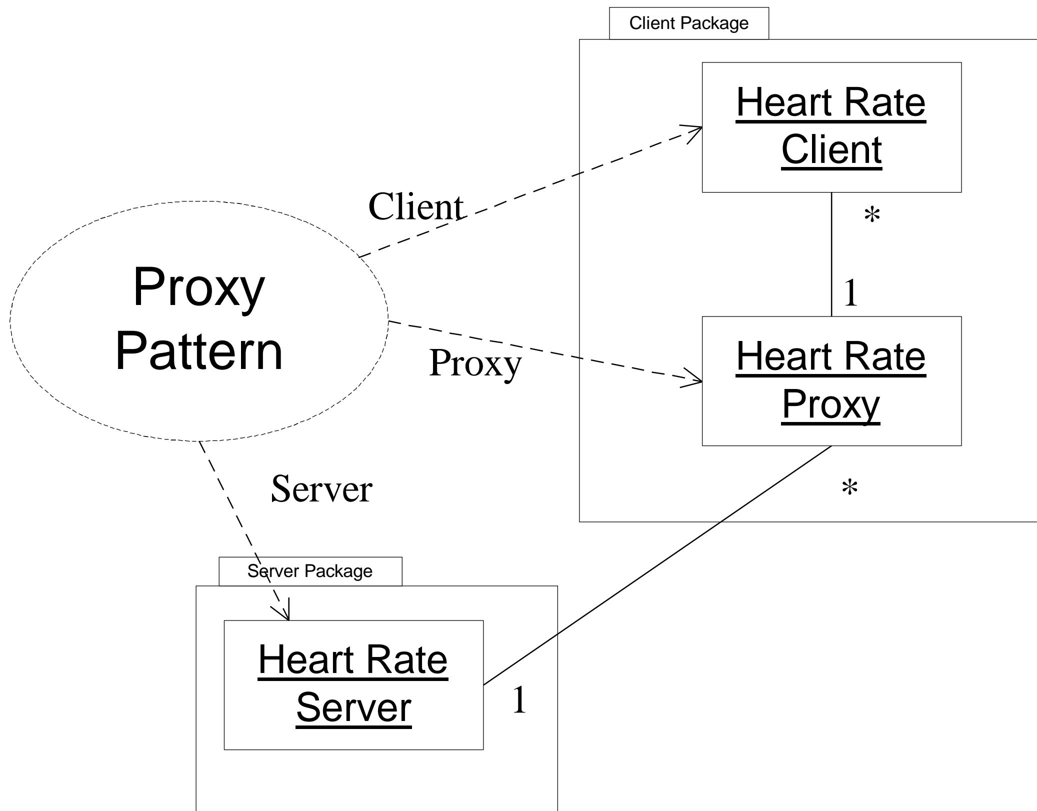


Figure 5: Proxy Pattern

Broker Pattern

The Broker Pattern is an elaborated Proxy Pattern which goes another step towards decoupling the clients from the servers. An object broker is an object which knows the location of other objects. The broker can have the knowledge *a priori* (at compile-time) or can gather the information dynamically as objects register themselves, or a combination of both. The primary advantage of the Broker Pattern is that it is possible to construct a Proxy Pattern when the location of the server isn't known when the system is compiled. This makes it particularly useful for systems using symmetric or semi-symmetric multiprocessing.

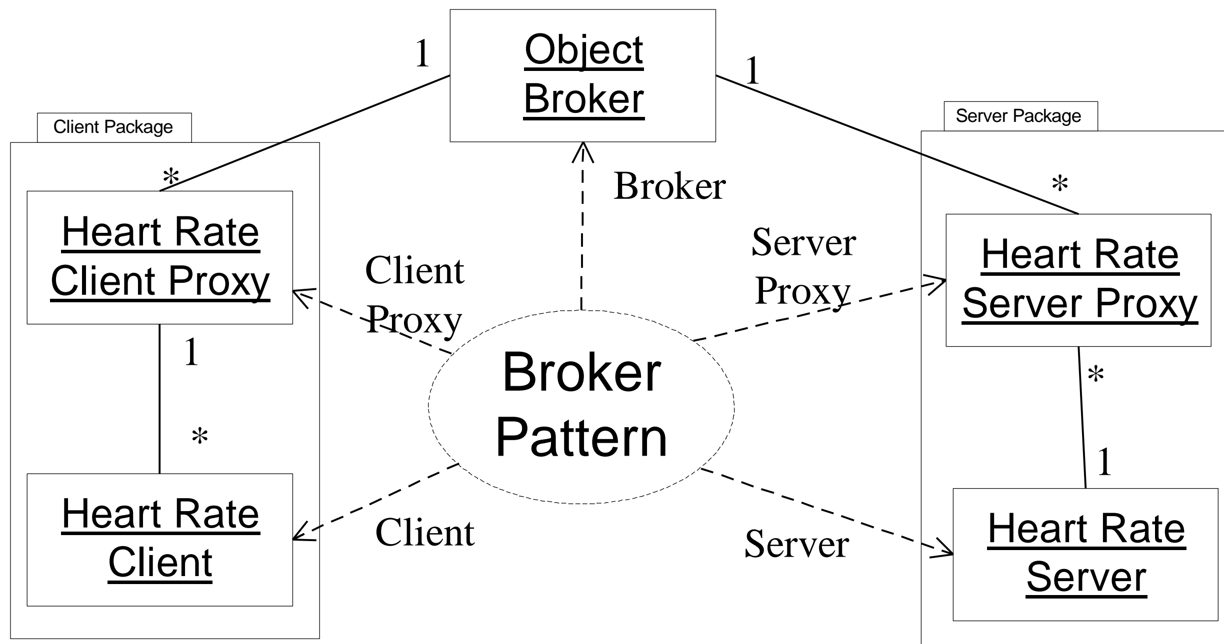


Figure 6: Broker Pattern

Asymmetric Multiprocessing

Most multiprocessing real-time systems dedicate processing tasks to particular processors. This arrangement works well, particularly for relatively simple systems. Load balancing in such systems must be determined at run-time. It is a relatively difficult procedure to add processors to an asymmetric multiprocessing system because the loading is determined at compile- or build-time.

Symmetric Multiprocessing

In a true symmetric multiprocessing system, when a task is spawned, an executive (usually the core of a distributed OS) determines which processor on which it should run. An advantage of symmetric multiprocessing systems is that the addition of processing node can make the software operate faster with no changes because the OS detects the new processor nodes and can dynamically load the new processors relieving the remaining processors of some of their workload. A final advantage is that this pattern can be used to dynamically rebalance processor load in the event of a processor node failure.

The disadvantages of this pattern include complexity in the OS, which translate into additional requirements for processor power and memory. Also many real-time systems are highly optimized and this pattern usually requires some additional levels of indirection. Finally, simple hardware devices must be interfaces with particular processors making it difficult to effectively use symmetry.

Semi-Symmetric Multiprocessing

The semi-symmetric multiprocessing (SSM) pattern is an optimization of the previous two patterns. It provides most of the benefits of the symmetric multiprocessing pattern while mitigating its drawbacks. Specifically, SSM does dynamic load balancing but only at boot time. This means that system speed can be enhanced by adding a processor node, but it is only recognized and balanced when the system is booted.

Resource Patterns

Many of the troubles of designing real-time embedded systems arise from the necessity of managing data resources. One of the problems is *heap fragmentation*. Fragmentation can arise when different sized blocks are allocated and released asynchronously from a heap. Overtime, the free space on the heap can fragment into small blocks. This can lead to allocation failures when a request is made which exceeds the size of the largest available block even though more than enough total memory may be available. The first two patterns, the *static allocation pattern* and the *fixed size allocation pattern*, address this particular problem.

Another problem occurs when sharing resources in multitasking environments. If a low-priority task locks a resource and a high priority task that needs that resource becomes ready to run, it is blocked from executing by the low priority task. This is called *priority inversion*. Worse, if the high priority task now suspends itself so that the low priority task can run, any task with an intermediate priority will preempt it, and indirectly preempt the waiting high priority task. This is called *unbounded priority inversion* and is a problem because it can lead to missed deadlines. The priority ceiling pattern offers a solution to this problem.

Static Allocation

One common pattern employed in embedded systems is to pre-allocate most or all objects in the system. If message objects are needed, some maximum number of them are created as the system starts up and then placed in a ring buffer. Because memory is never released, heap fragmentation cannot occur. Also, overhead is minimized during run-time because time need not be taken to call object constructors. In certainly languages (notably C++) care must be taken to avoid inadvertent calls to the constructor (such as the copy constructor), but this is true of embedded systems in general. This pattern is similar to the Factory Pattern in [9].

One problem with this pattern is that it may not scale up well to large problems. It may be impossible to pre-allocate all possible needed objects because of a lack of total memory. Many systems can operate in a much smaller amount of total memory if they can dynamically create and destroy objects as needed.

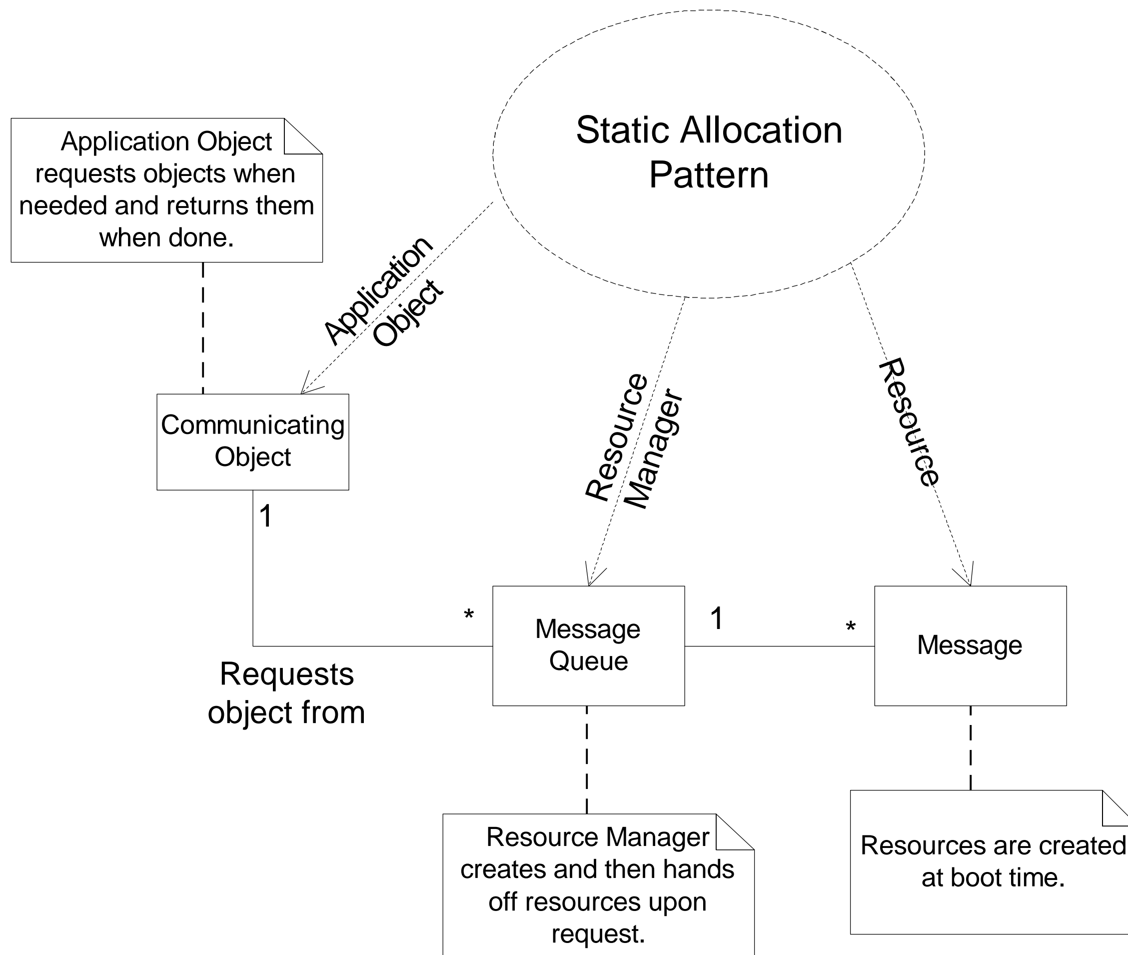


Figure 7: Static Allocation Pattern

Fixed Size Allocation

The Fixed Sized Allocation Pattern offers a different solution to the problem of heap fragmentation. Heap fragmentation occurs because memory blocks of different sizes are created. When they are destroyed, they leave holes in the memory space. Because these holes are of different size than the requests, the holes may be unusable later. Fixed size allocation solves this problem by creating a heap from which only a single sized block may be allocated. If the needed object is smaller than the block, then the unused portion of the allocated block is "wasted."

It is not uncommon to provide a number of different block sizes in different heaps in an effort to minimize the waste. A first-fit algorithm is then employed to find the smallest block that meets the need.

This pattern eliminate fragmentation, which can be a severe disability in systems that must operate for long periods of time between resets, such as monitoring equipment and space

probes. It can be somewhat wasteful of memory because each allocated memory block will typically have at least some memory that is not being used.

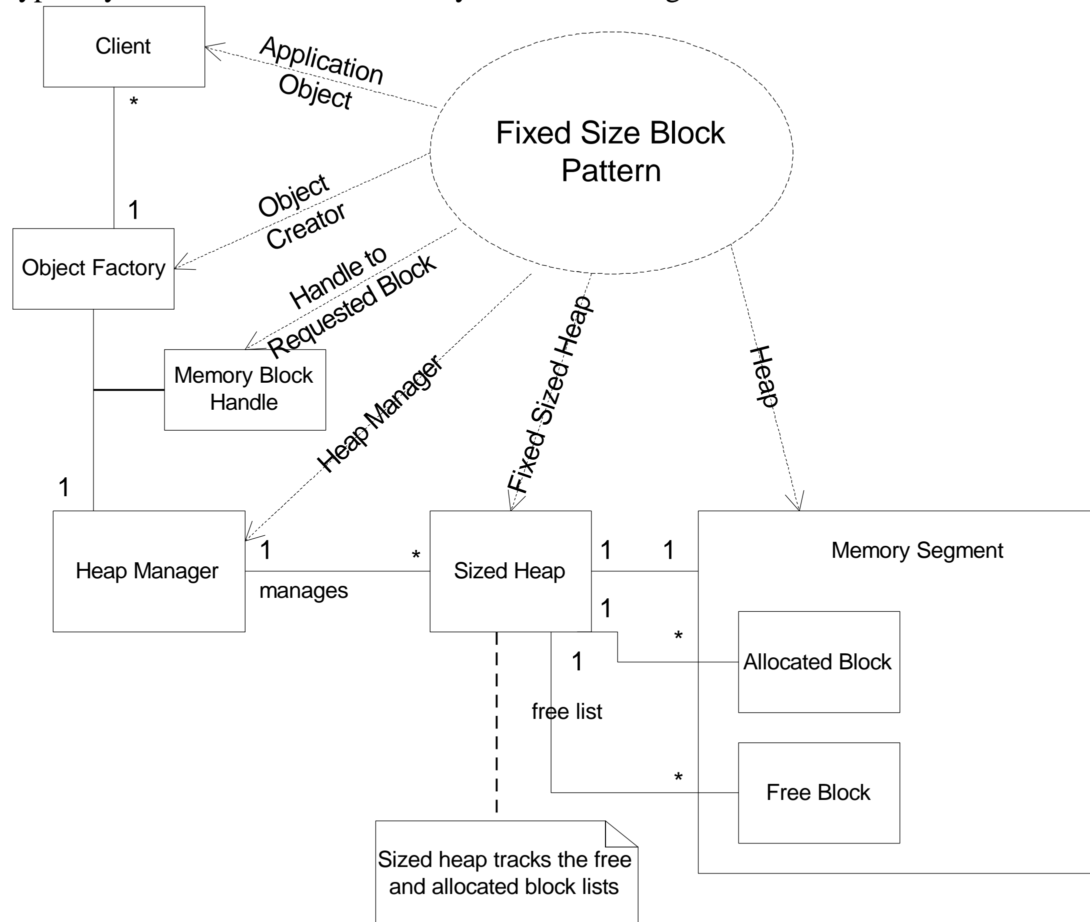


Figure 8: Fixed Sized Block Pattern

Priority Ceiling

If life was simple, concurrent tasks would never have to share resources directly with other tasks[5]. However, life is anything but simple, so tasks operating in independent concurrent threads must rendezvous and synchronize every so often. However, this is itself problematic. In order to eliminate the possibility of data corruption due to simultaneous read and write accesses, some means must be provided to allow only a single object access to a resource at any given time. Monitors and semaphores are common such means – they both work by granting exclusive access to a single object and blocking other requests. This means that if a low priority task locks a resource needed by a high priority task, the high priority task must block itself and allow the low priority task to execute, at least long enough to release the needed resource.

[5] On the other hand, if life was simple you could be replaced by a pimply-faced teenager whose job is now filled with phrases like "Ya'll want fries with that?"

The execution of a low priority task when a higher priority task is ready to run is called *priority inversion*. The naïve implementation of semaphores and monitors allows the low priority task to be interrupted by higher priority tasks that do not need the resource. Because this preemption can occur arbitrarily deep, the priority inversion is said to be *unbounded*. It is impossible to avoid at least one level of priority inversion in multitasking systems that must share resources, but one would like to at least bound the level of inversion. This is problem addressed by the *priority ceiling pattern*.

The basic idea of the priority ceiling pattern is that each resource has an attribute called its priority ceiling. The value of this attribute is the highest priority of any task that could ever use that particular resource. The active objects[6] have two related attributes: nominal priority and current priority. The nominal priority is the normal executing priority of the task. The object's current priority is changed to the priority ceiling of a resources it has currently locked as long as the latter is higher.

The advantage of this pattern can be illustrated by considering three tasks scheduled without the use of the priority ceiling, as shown in Figure 9. Point B indicates the point at which Task 2 preempts the lowest priority, Task 3. However, if the priority ceiling pattern is used, then as soon as Task 3 locks the resource, its priority is elevated to the priority ceiling of the resource (which the priority of Task 1). Therefore, at Point B (where Task 2 becomes ready run), Task 3's current priority is higher than that of Task 2, so Task 3 is not preempted by Task 2. Once Task 3 is done with the resource, it's priority is restored to its nominal priority and then the highest priority waiting task (in this case, that's Task 1) preempts Task 3 and runs.

[6] Which, in the UML, are the root objects of threads.

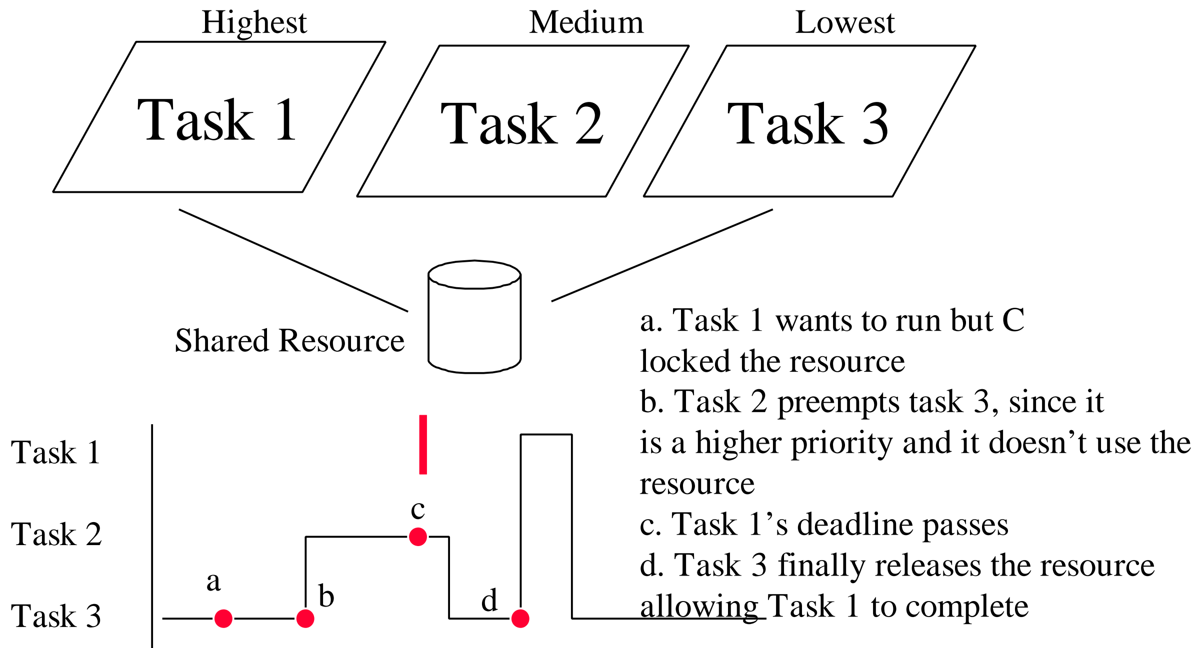


Figure 9: Priority Ceiling Pattern In Action

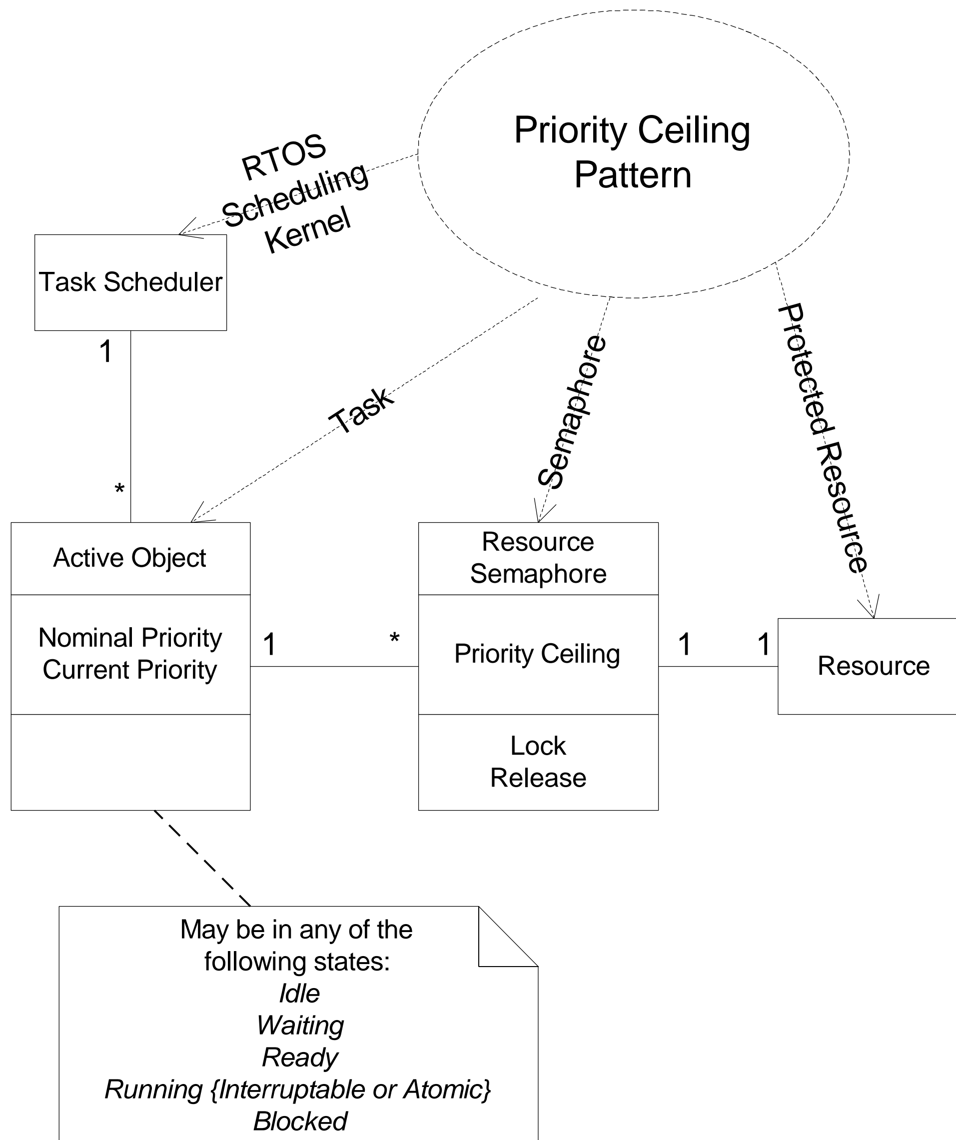


Figure 10: Priority Ceiling Pattern

Safety and Reliability Patterns

There are many architectural patterns that relate to improving safety and reliability. A detailed discussion of safety and reliability is beyond the scope of this paper. The interested reader is referred to [1], [2], [13], and [14]. In principle, though, they all require

- some form of redundancy
- some mechanism to detect faults using the redundancy
- some corrective action to be taken in the presence of faults

Homogeneous Redundancy

A common approach to improvement of reliability and safety is through multiple identical channels. In this context, a *channel* is a collaboration of objects which performs a series of safety-relevant computations or actuations. The use of identical channels allows protection from random faults, such as component failures, but not against systematic faults, such as software errors. Homogenous redundancy can be used in parallel using a voting scheme. It can also be used as a switch-over backup in the event of a detected failure in the currently executing channel.

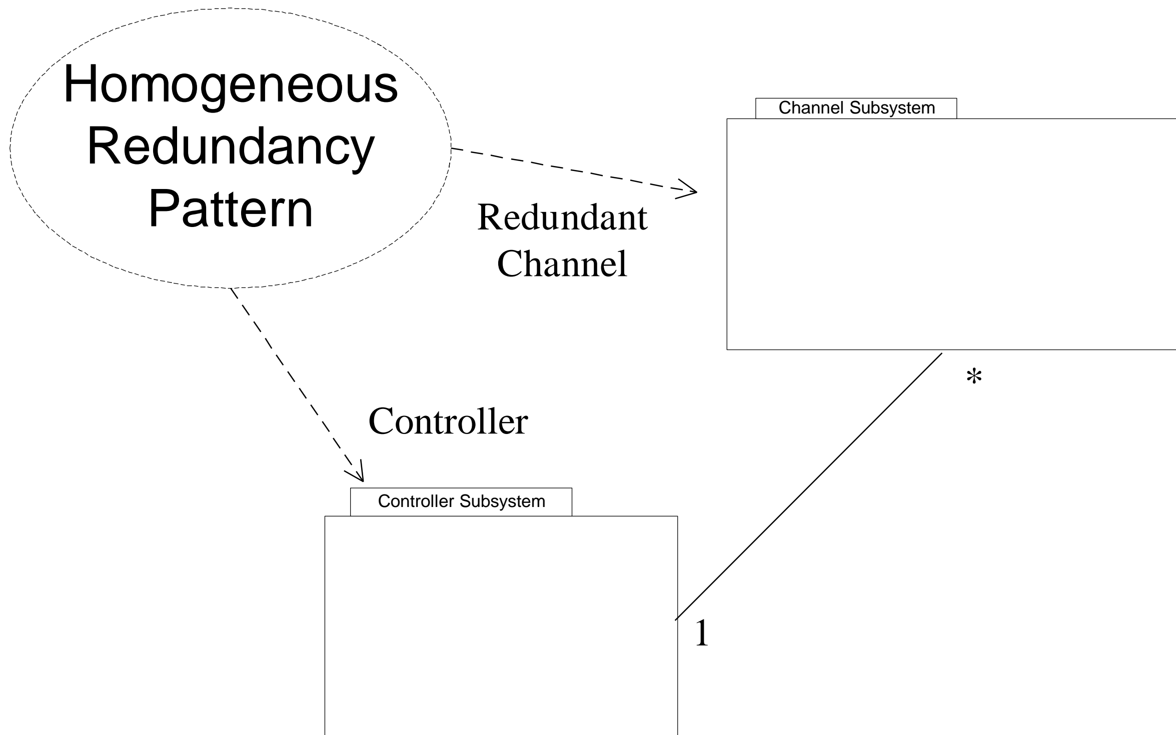


Figure 11: Homogeneous Redundancy Pattern

An advantage of this pattern is the low R&D cost – since there is only a single channel to design. It's primary disadvantage is the lack of coverage for systematic faults and increase deployment costs over non-redundant systems.

Heterogeneous Redundancy

Another common pattern is the *heterogeneous redundancy pattern* (a.k.a. diverse redundancy pattern). This is similar to the previous pattern except that the channels are differently implemented: the hardware is different and the software is typically built using a different design and executable code.

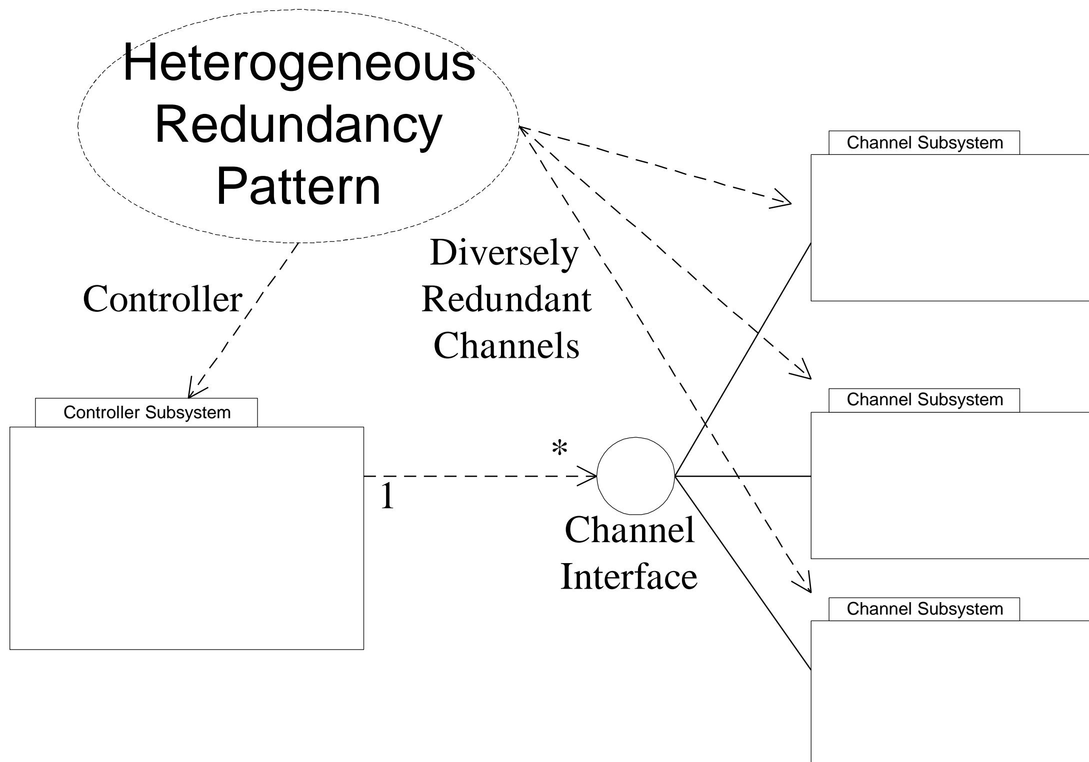


Figure 12: Heterogeneous Redundancy Pattern

The heterogeneous redundancy pattern covers systematic as well as random faults, but at an increase R&D effort and cost as well as increased deployment costs.

Sanity Check

The cost for fully redundant system may be prohibited in some cases even though some protection from faults may be required. The sanity check pattern uses a primary heavyweight channel plus a second lightweight channel. This latter channel cannot take over the full duties of the primary channel but can detect faults within it. The advantage of the sanity check pattern is that some protection against faults is provided at a lower cost than either of the two previous patterns. The disadvantage is that inadequate recovery means may be provided in the event of a failure. This pattern is particularly useful when there is a known fail-safe state.

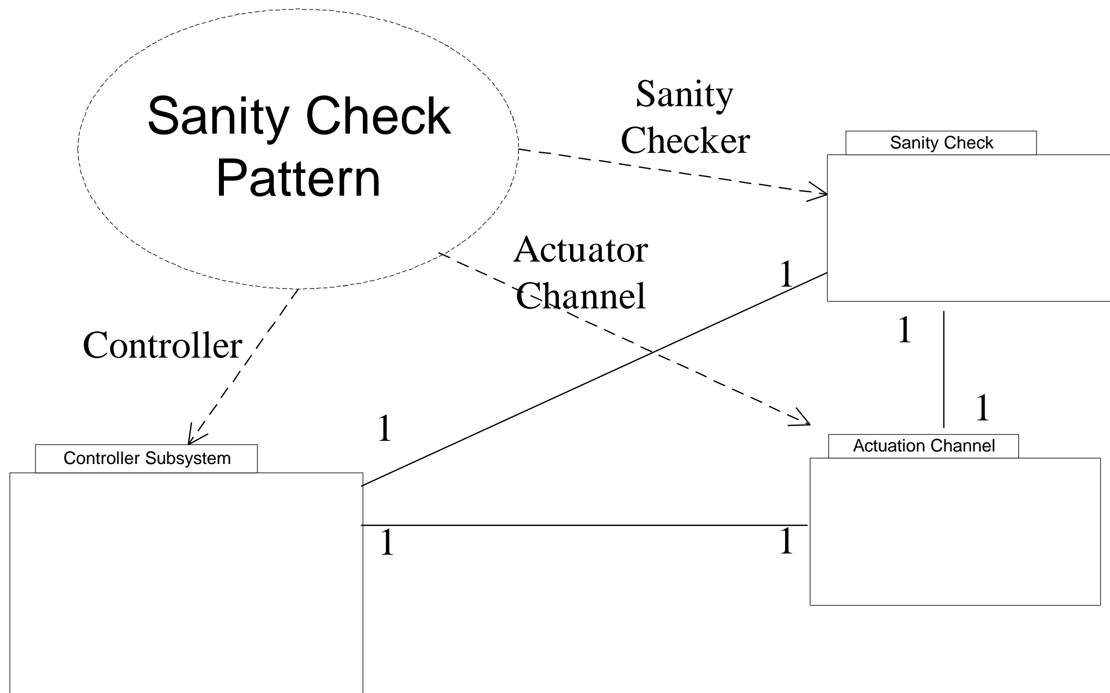


Figure 13: Sanity Check Pattern

Monitor-Actuator

A common type of the sanity check pattern is the monitor-actuator pattern. In this latter pattern, the monitor channel does not check the computations of the primary channel directly as in the former pattern, but instead checks on the results of the actuation of the primary channel. It is important that this monitoring is provided by a separate set of sensors than is used by the actuation channel to provide protection from common-mode faults (faults that occur in more than one channel).

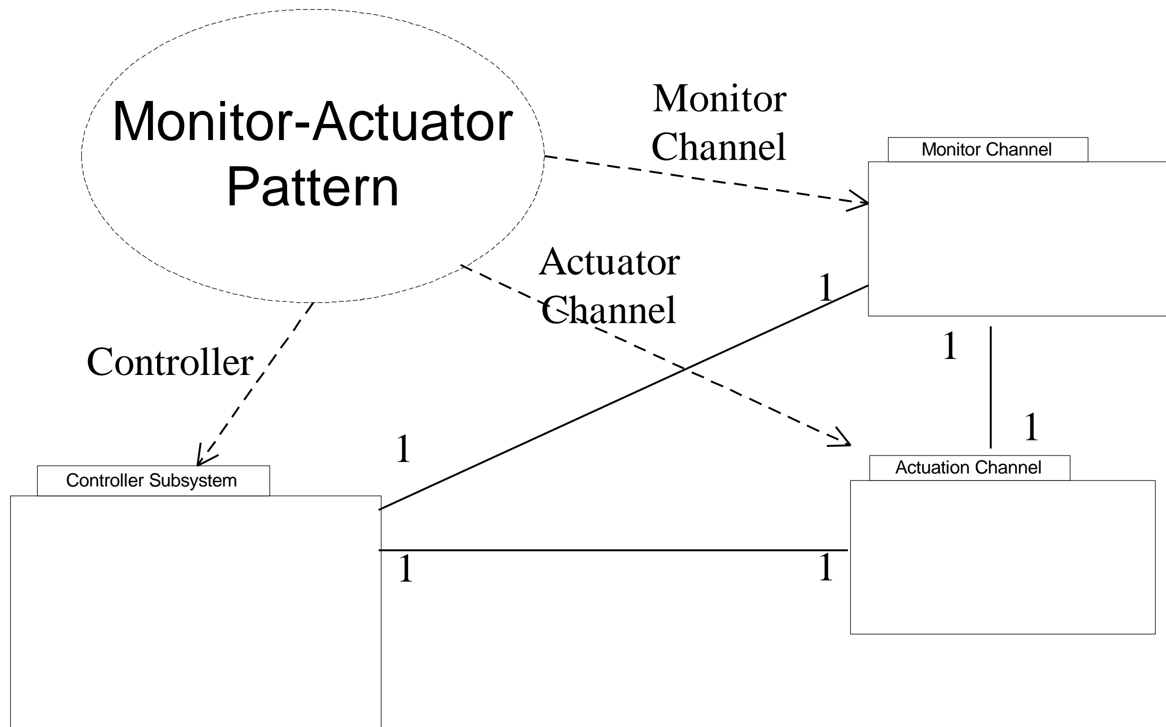


Figure 14: Monitor-Actuator Pattern

The monitor-actuator pattern provides the low-cost redundancy of the sanity check pattern but relies on external sensors to achieve the fault identification. Thus, the identification of faults will not be as fine-grained as other patterns, but on the other hand, is based on real-world data.

Watchdog

Another variant of the sanity check pattern is the watchdog pattern. In the watchdog pattern, an object called a watchdog is signaled periodically by various objects in the system. The watchdog pattern is very widely used with additional hardware support (such as an independent timebase) to provide isolation from CPU faults.

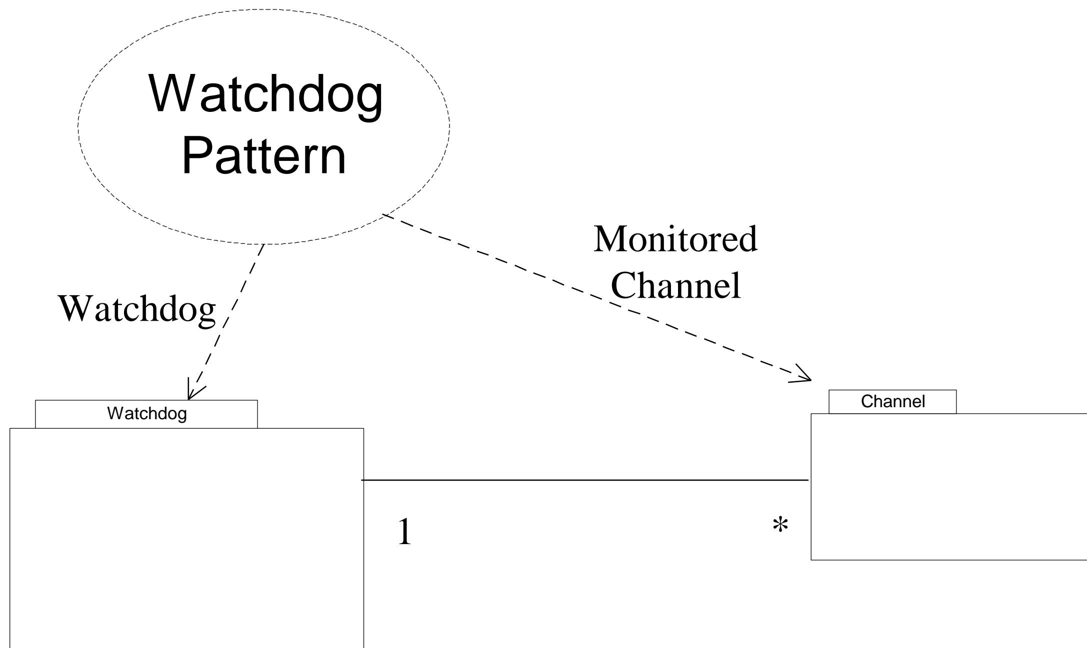


Figure 15: Watchdog Pattern

The watchdog pattern is a low cost solution but it has limited applicability. Specifically, it determines the health of the system from the fact that it is receiving events within a specified time window. This does detect hung system faults but not other kinds of faults. More complex watchdogs which must receive computed keys in a particular sequence are also in common use, but the coverage of possible faults is minimal.

Safety Executive

The final architectural pattern is the safety executive pattern. This pattern is used when many multiple safety concerns are addressed by a single system, fault detection is complex, or the fault recovery mechanisms are elaborate.

The advantage is the safety executive pattern is that the safety mechanisms for complex systems are centralized and isolated away from other system components. This pattern also has the ability to handle complex fault isolation and recovery. Disadvantage is the inherent complexity and effort required in constructing an effective safety executive.

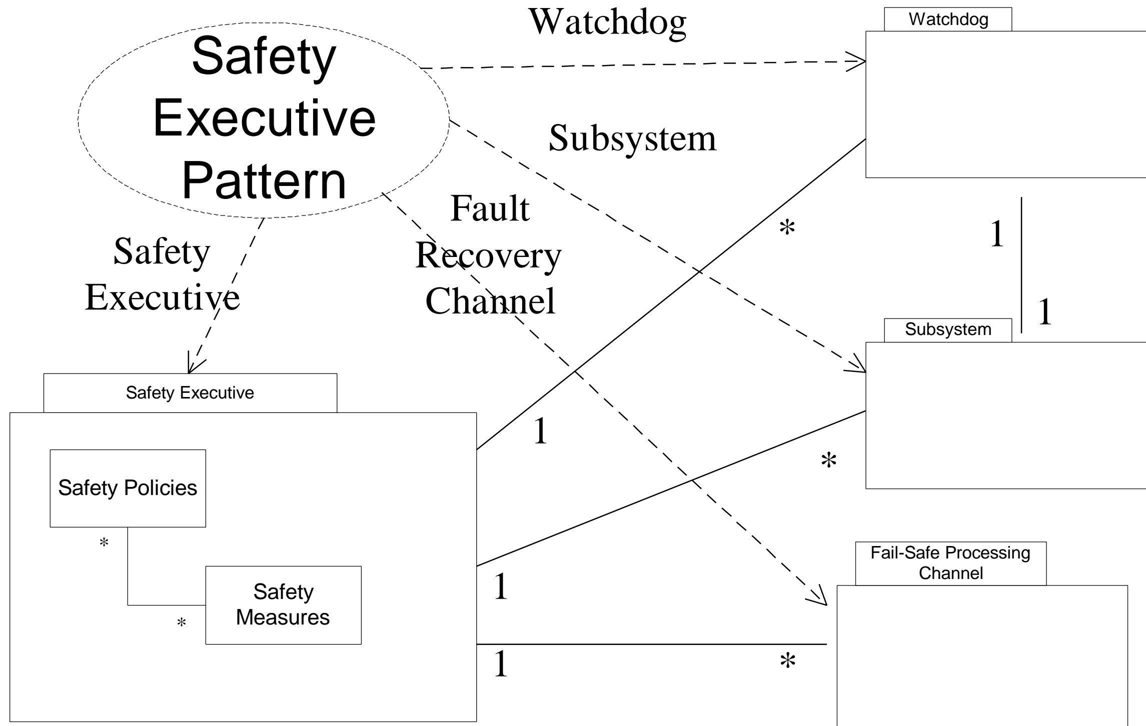


Figure 16: Safety Executive Pattern

Mechanistic Design Patterns

Mechanistic design patterns are smaller in scope from their architectural brethren. Such patterns typically involve two to dozen classes.

Category	Pattern Name	Purpose
Simple Patterns	Observer	Allow multiple clients to effectively share a server and be autonomously updated
	Transaction	Control communication between objects with various levels of reliability
	Smart Pointer	Avoid problems associated with dumb pointers
Reuse	Container	Abstract away data structuring concepts from application domain classes to simply model and facilitate reuse
	Interface	Abstract away the type of an object from its implementation to support multiple implementations of a given type and to support multiple types with a common internal structure.
	Policy	Provide the ability to easily change algorithms and procedures dynamically.
	Rendezvous	Provide a flexible mechanism for light-weight intertask communication.

State Behavior	State	Provide an optimal state machine implementation when some state changes are infrequent
	State Table	Provide an efficient means to maintain and execute large complex state machines

Simple Patterns

Observer

It is common that a single source of information acts as a server for multiple clients who must be autonomously updated when the data value changes. This is particularly true with real-time data acquired through sensors. The problem is how to design an efficient means for all clients to be notified.

The Observer pattern[7] is one design solution. A single object, called the *server*, provides the data automatically to its clients, called *observers*. These are abstract classes which may be subclassed (into *Concrete Server* and *Concrete Observer*) to add the specialized behavior to deal with the specific information being served up.

The observers register with the server by calling the server's *Subscribe()* method and deregister by calling the *Detach()* method. When the server receives a subscribe message, it creates a Notification Handle object which includes the address of the object. This address may be either a pointer, if the object is in the same data address space, or a logical address or identifier to be resolved by a separate communications subsystem or object broker if the target object is in a remote address space.

The Notification Handle class is subclassed to distinguish its update policy. The update policy defines the criteria for when data is sent to the observer. Typically, this is periodic, episodic, or epi-periodic (both). In some cases, it may be sufficient to use the same policy universally, but using a policy makes the design pattern more general. It is very common for an episodic policy to be used exclusively, but this is insufficient for many applications. In safety critical systems, for example, a lost message could result in an unsafe system. By periodically sending the data to the observers, the system is hardened against random message loss.

[7] A.K.A. *Publish-Subscribe*

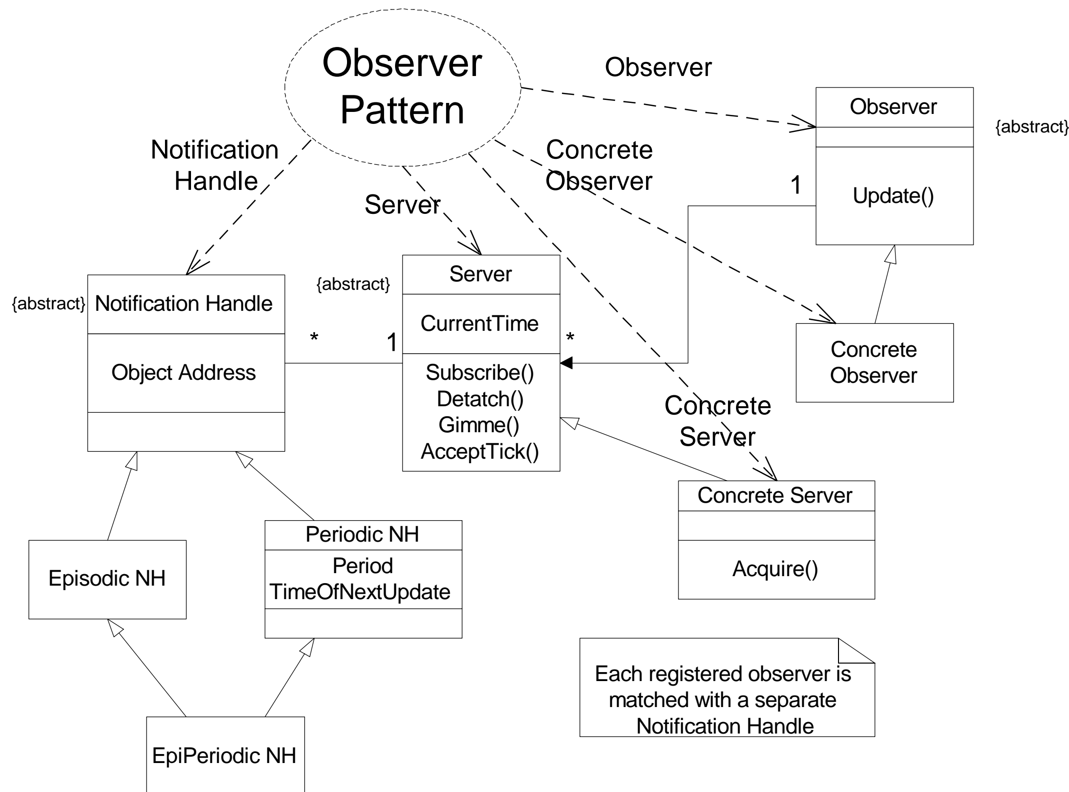


Figure 17: Observer Pattern

Transaction

Real-time systems use communication protocols to send and receive critical information, both among internal processors, and with external actors in the environment. Within the same system, different messages may have different levels of criticality, and so may have different requirements for the reliability of message transfer. Further, different media have different reliability as do different environments.

The transaction is used when reliable communications is required over unreliable media or when extraordinary reliability is required. For example, a system needs might need three distinct levels of communications reliability:

1. At Most Once (AMO) – a message is transmitted only once. If the message is lost or corrupted, it is lost. This is used when lightweight transfer is required and the reliability of message transfer is high compared to the probability of message loss.
2. At Least Once (ALO) – a message is transmitted repeatedly until either an explicit acknowledgement is received by the sender or a maximum retry count is exceeded. This is used when the reliability of message transfer is relatively low compared to the probability of message loss, but receipt of the same message multiple times is ok.

3. Exactly Once (EO) – a message is treated as an ALO transaction except that should a message be received more than once due to retries, only the first message instance will be acted upon. This is used when message transfer reliability is relatively low but it is important that a message is acted on only once. *Increment* or *toggle* messages, for example, must only be acted on once.

The Transaction Pattern is particularly suited to real-time systems which use a general communication protocol with a rich grammar. It allows the application designers flexibility in their choice of communications method so that they may optimize for speed or reliability.

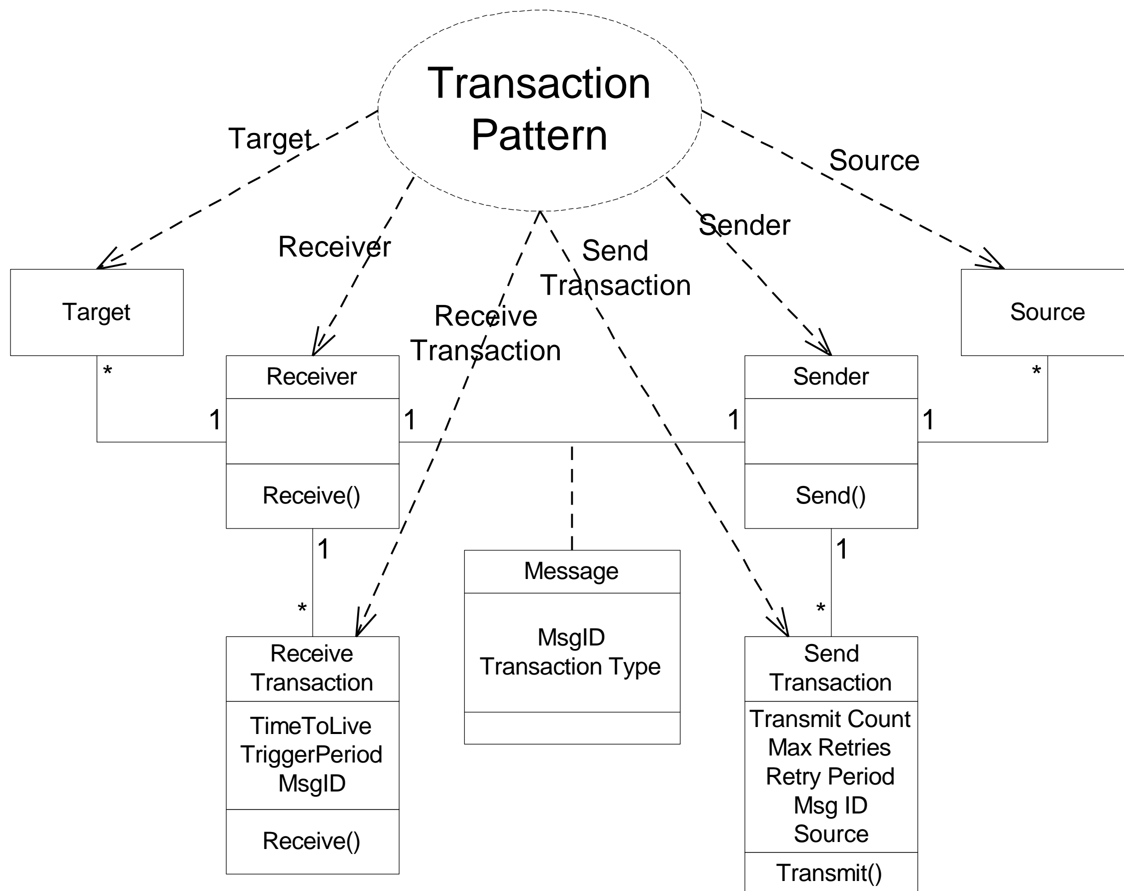


Figure 18: Transaction Pattern

Smart Pointer

The smart pointer is a common pattern meant to eliminate, or at least mitigate the myriad of problems that stem from the manual use of raw pointers:

- While raw pointers have no constructor to leave them in a valid initial state, smart pointers can use their constructors to initialize them to NULL or force the precondition that they are constructed pointing to a valid target object.
- While raw pointers have no destructor and so may not deallocate memory if they suddenly go out of scope, smart pointers are determine whether or not it is appropriate to deallocate memory when they go out of scope and call the delete operator
- While a raw pointer to a deleted object still holds the address of the memory where the object used to be (and hence can be used to reference that memory illegally), smart pointers can automatically detect that condition and refuse access.

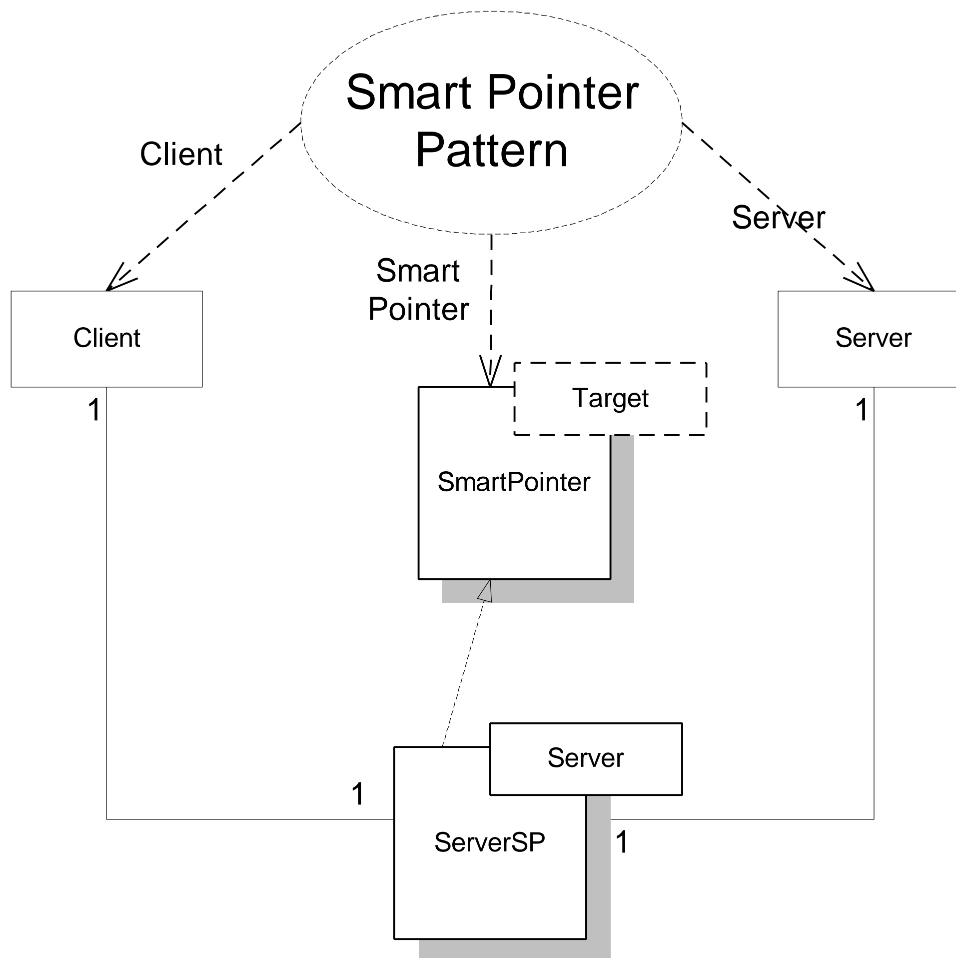


Figure 19: Smart Pointer Pattern

Internal to the smart pointer is a (static) reference count, which tracks the number of current clients to the server. This count is incremented when smart pointers to the same object are added and decremented when smart pointers are deleted. When the last smart pointer is being deleted, the memory to the object is released by the smart pointer prior to its destruction.

Reuse

Container

Analysis models the "what" of a system – what it is, what are the fundamental concepts involved and what are the important relations and associations among them. Design specifies the "how" of all the unspecified portions of the system. One of the important "hows" of design is how each and every object-object message will be implemented; is it a function call, an OS mail message, a bus message, or something even more exotic? Another important "how" is the resolution of associations with multi-valued roles.

When one object has a 1-to-many association, the question arises as to the exact mechanism the "1" class will use to access the "many" objects. One solution is to build features into the "1" class to manage the set of contained objects. These facilities typically manifest themselves as operations such as add(), remove(), first(), next(), last(), and find(). Often the semantics of the associate dictate elaborate operations, such as maintaining the set of objects in a specific order or balancing the tree. The common solution to these problems is to insert a container object (a.k.a. *collection object*) between the "1" and the "many."

Adding a container object to manage the aggregated objects doesn't solve the entire problem because often the container must be accessed from several different clients. If the container itself keeps track of the client position, then it will become confused in a multi-client environment. To get around this, *iterators* are used in conjunction with the containers. An iterator keeps track of where the client is in the container. Different clients use different iterators so that the separate concerns of managing the collection and tracking position within the container are abstracted away from each other. A single client may use several different iterators. The Standard Template Library (STL), a part of the ANSI C++ standard, provides many different containers with a variety of iterators, such as *first*, *last*, and so on.

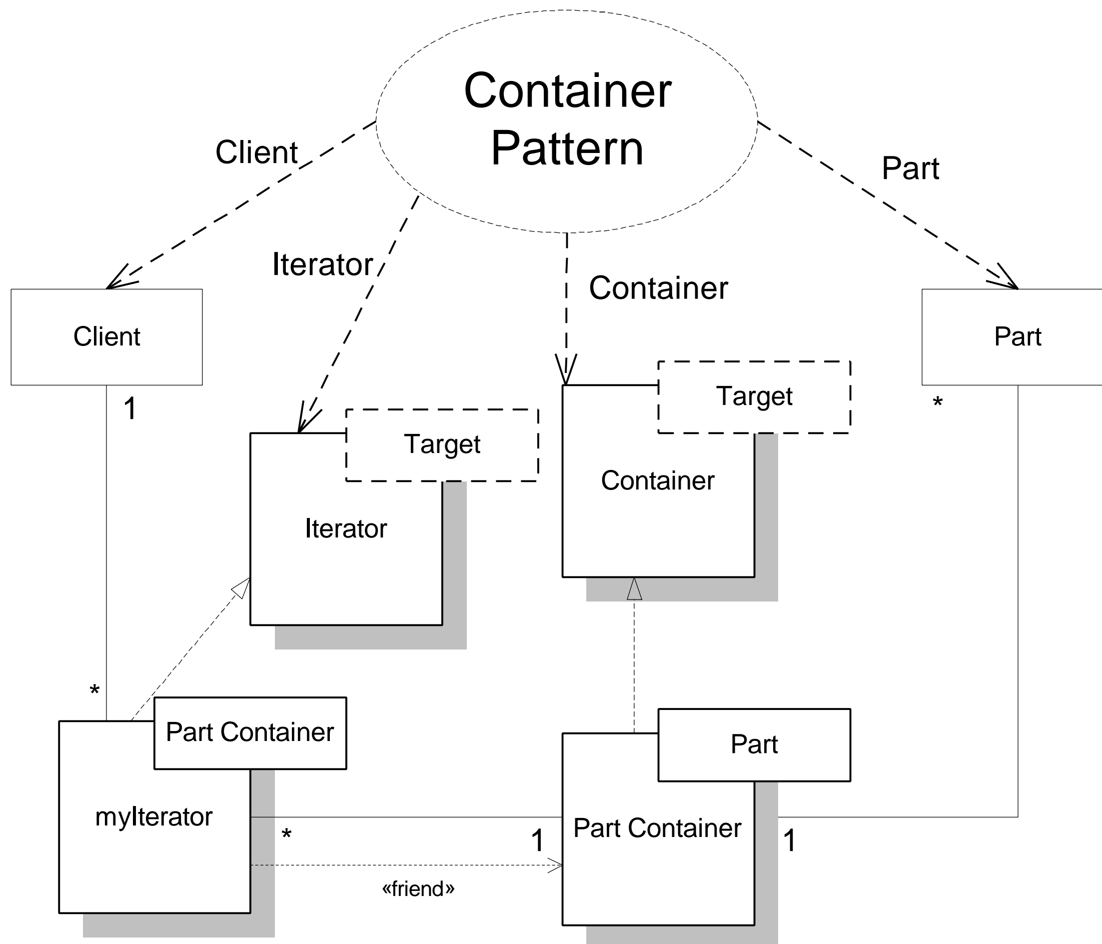


Figure 20: Container Pattern

Interface Pattern

In languages like C++, the interface provided by a class is bound tightly to its implementation. As discussed previously, strictly speaking the interface of an object is its *type* and the implementation specification is its *class*. C++ mixes these metaphors so that most of the time the difference is unnoticeable. Unfortunately, binding the type and class together limits the reusability of a class. There are a number of cases in which explicit separation of interface and implementation is useful.

First, a common implementation may be appropriate for a variety of uses. If a class could provide different interfaces, a single underlying implementation could meet several needs. For example, many common computer science structures, such as trees, queues, and stacks, can actually use a single underlying implementation, such as a linked list. The relatively complex innards of the common implementation can be used in different ways to implement the desired behavior, even though the ultimate clients are clueless as to what happens behind the scenes.

Secondly, by separating an interface, it becomes easier to change an implementation or add a new and different interface. As in our container example in the previous paragraph, it becomes a simple matter to add a new container (for example, an extensible vector) by creating an interface that provides the correct services to the client but implements those services in terms of primitive operations of existing containers.

Lastly, it happens sometimes that you want different levels of access into the internals of an object. Different interfaces can be constructed to provided different levels of access for different client objects and environments. For example, you might want classes providing services to users be able to use only a subset of all operations, while a different set be provided in "service mode" and a much different set provided in "remote debugging mode."

The Interface Pattern[8] solves all of these problems. It is a very simple pattern but is so common that UML actually provides the stereotype «type» in the language specification.

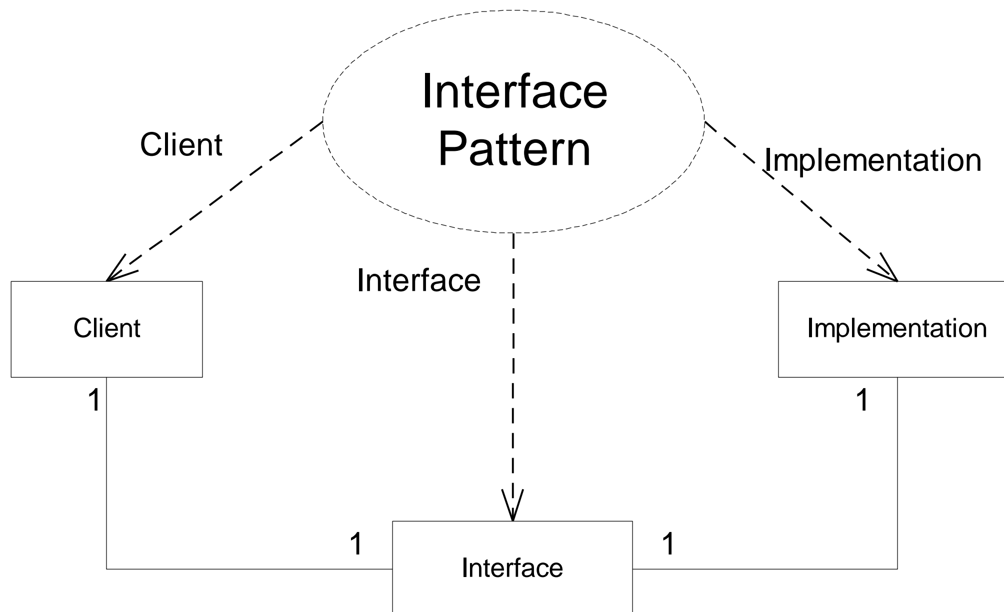


Figure 21: Interface Pattern

Policy

Often classes are structurally similar or even identical, but differ in terms of how they operate internally. For example, it is possible that a class looks the same but makes different time/space/complexity/safety/reliability optimization choices. The selection of different algorithms to implement the same black box behavior is called a *policy*. Policies can be abstracted away from the main class to simplify the interface, improve reuse, and even allow dynamic choices of policies based on operating condition or state.

[8] a.k.a., the Adapter Pattern

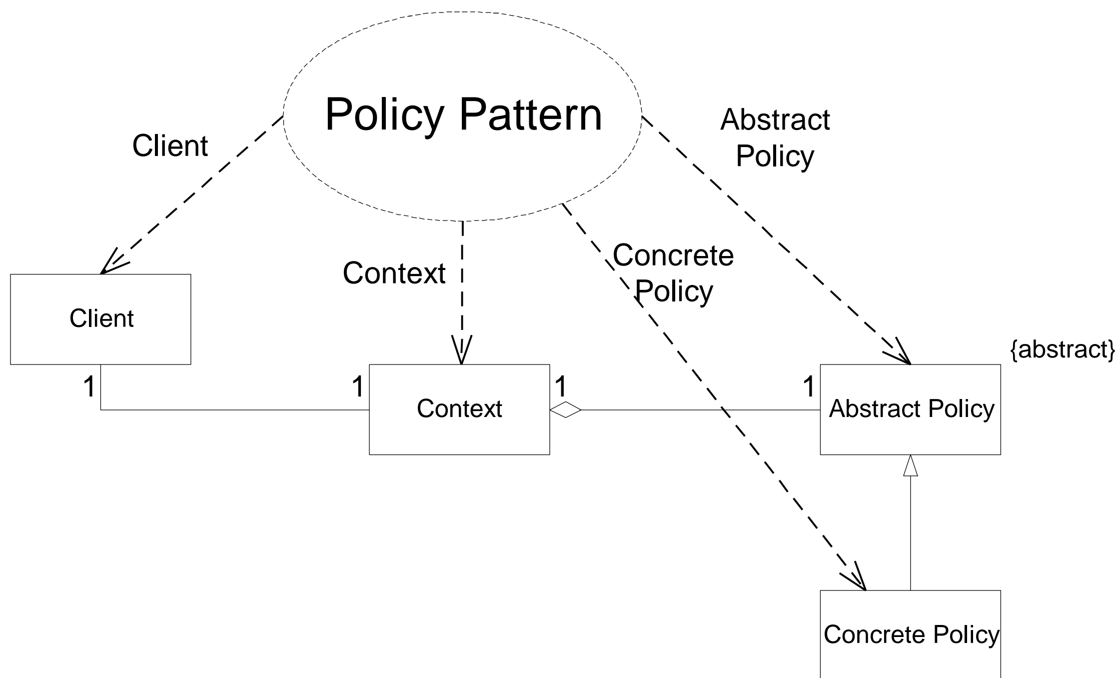


Figure 22: Policy Pattern

Rendezvous

A rendezvous refers to the synchronization of concurrent tasks. In fact, the use of a mutual exclusion semaphore is also called a *unilateral rendezvous*. The more common use of the term rendezvous refers to the synchronization of more than one task. For example, the rendezvous of more than two tasks is referred to as a bilateral rendezvous.

This pattern consists of a coordinating passive object (called the rendezvous object), clients which must synchronize, and mutex blocking semaphores. Blocking semaphores are used to force the threads to waiting until all preconditions are met. More elaborate behavior can be implemented using timed or balking semaphores, if desired.

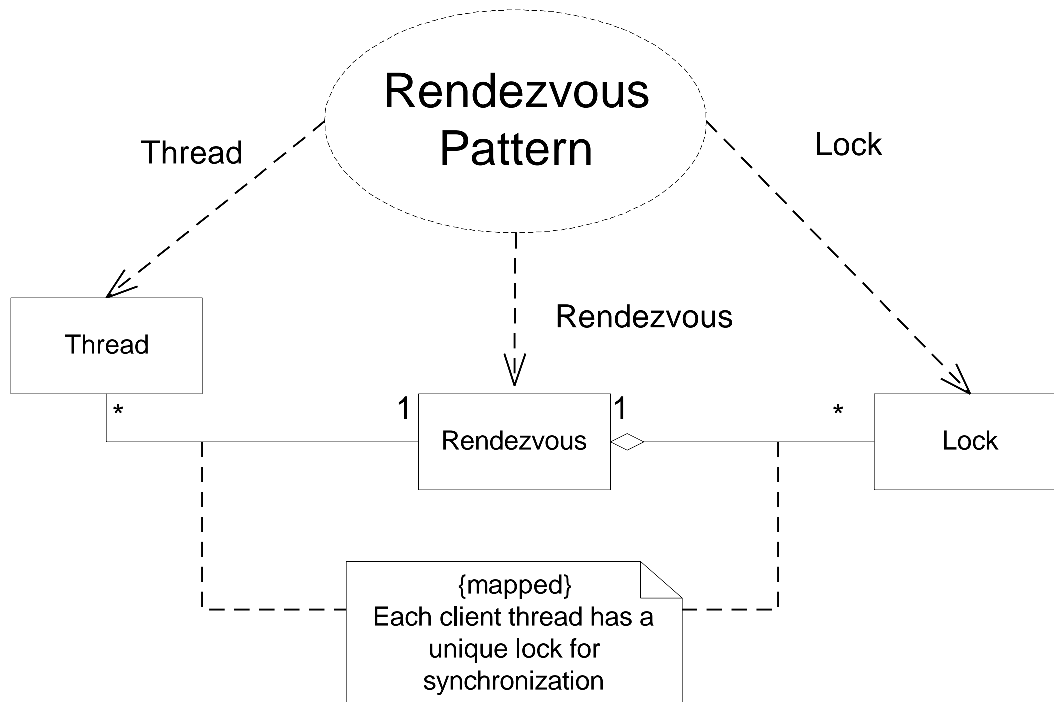


Figure 23: Rendezvous Pattern

State Behavior

State

Many systems spend most of their time in only a few states. For such systems, it is more efficient to have a heavyweight process for transitioning to the less-used state if it can make the more-used state transitions lighter weight. This pattern also facilitates reuse in subclasses because often subclasses only change a small set of states. If the state pattern is used, then only the state classes that are changed need to be modified. Thus the changes are better encapsulated, simplification polymorphic behavior of reactive classes.

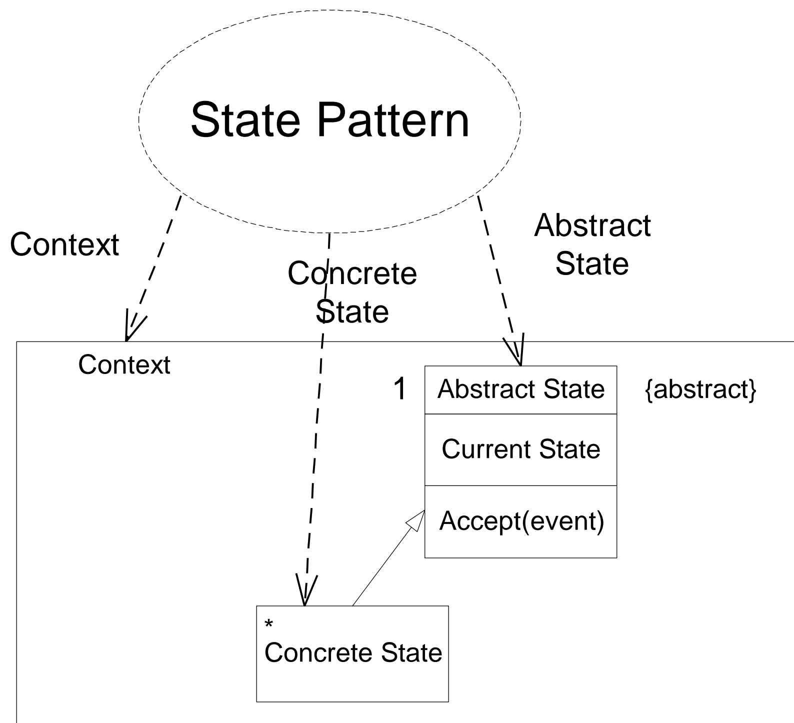


Figure 24: State Pattern

State Table

The State Table Pattern provides a simple mechanism for managing state machines with an efficiency of O(c), where c is a constant. This is a preferred mechanism for very large state spaces because the time to handle a transition is a constant (not including the time to execute actions associated with state entry or exit, or the transition itself). Another advantage is that this pattern maps directly to tabular state specifications, such as those used to develop many safety-critical systems.

The State Table pattern hinges upon the state table. This structure is typically implemented as an n x m array, where n is the number of states and m is the number of transitions. Each cell contains a single pointer to a Transition object which "handles" the event with an *accept* operation. The operation returns the resulting state. Both events and states are represented as an enumerated type. These enumerated types are used as indices into the table to optimize performance. Although this pattern has a relatively high initialization cost, its execution cost is low after it is set up.

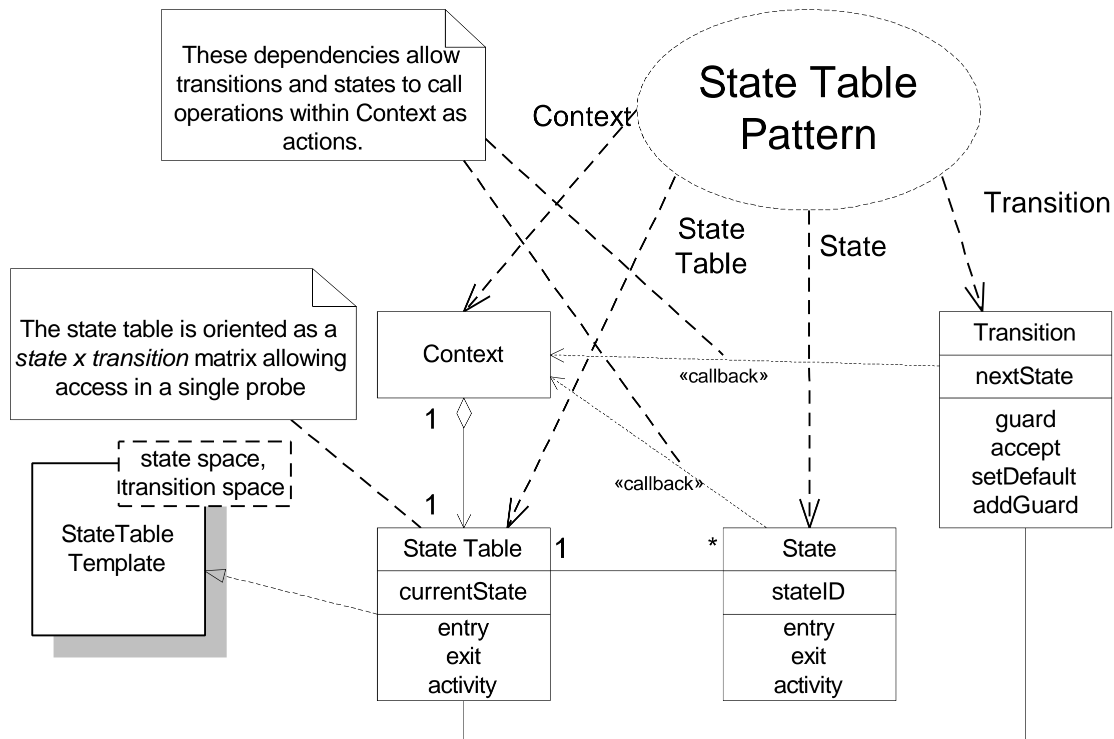


Figure 25: State Table Pattern

References

[1] Douglass, Bruce Powel *Real-Time UML: Efficient Objects for Embedded Systems*, Addison-Wesley-Longman, 1998

[2] Douglass, Bruce Powel *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications*, Reading, MA: Addison-Wesley-Longman, Spring, 1998

[3] Fowler, Martin and Kendall Scott *UML Distilled: Applying the Standard Object Modeling Language* Reading, MA: Addison-Wesley-Longman, 1997

[4] *UML Summary* Version 1.1 September, 1997. Rational Corp, et. al. As submitted to the OMG.

[5] *UML Semantics* Version 1.1 September, 1997. Rational Corp, et. al. As submitted to the OMG.

[6] *UML Notation Guide* Version 1.1 September, 1997. Rational Corp, et. al. As submitted to the OMG.

[7] *A System of Patterns: Pattern-Oriented Software Architecture* Buschmann, et. al., John Wiley & Sons, 1996

[8] *A Pattern Language* Alexander, Ishikawa, and Silverstain, Oxford University Press, 1977

[9] *Design Patterns* by Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995

[10] *Pattern Languages of Program Design* ed. Coplien and Schmidt, Addison-Wesley, 1995

[11] *Pattern Languages of Program Design 2* ed. Vlissides, Coplien and Kerth, Addison-Wesley, 1996

[12] *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems* by Klein, Ralya, Pollak, Obenza, and Harbour, Kluwer Academic Publishers, 1993.

[13] *Safety-Critical Computer Systems* by Neil Storey, Addison Wesley, 1996.

[14] *Safety Critical Systems* by Bruce Powel Douglass, Embedded Systems Conference – Spring, 1998.

About I-Logix

I-Logix Inc. is a leading provider of application development tools and methodologies that automate the development of real-time embedded systems. The escalating power and declining prices of microprocessors have fueled a dramatic increase in the functionality and complexity of embedded systems—a trend which is driving developers to seek ways of automating the traditionally manual process of designing and developing their software. I-Logix, with its award-winning products, is exploiting this paradigm shift.

I-Logix technology supports the entire design flow, from concept to code, through an iterative approach that links every facet of the design process, including behavior validation and automatic "production quality" code generation. I-Logix solutions enable users to accelerate new product development, increase design competitiveness, and generate quantifiable time and cost savings. I-Logix is headquartered in Andover, Massachusetts, with sales and support locations throughout North America, Europe, and the Far East. I-Logix can be found on the Internet at http://www.ilogix.com