# Caterpillar's Fate:
# A Pattern Language for
# Transformation from Analysis to Design

Norman L. Kerth
P.O. Box 2205
Beaverton, OR 97075
(503) 297-8677
Internet: 72073.3222@compuserve.com

## 1.0 Abstract

*Caterpillar's Fate* is a pattern language used to support the transformation from fine analysis documents to an initial software design. Just as the concept *metamorphosis* is used to explain the magical emergence of butterflies; Caterpillar's Fate explores the magic of constructing a system of objects from an object-free analysis.

## 2.0 Introduction

*Caterpillar's Fate* is an odd name for a software engineering activity. Nevertheless, the name is a most accurate description of the role that this pattern language play's in the software life cycle. A caterpillar's fate is *metamorphosis* -- the seemingly magical transformation from a sedentary creature to an aerial one. The Caterpillar's Fate pattern language is used to guide the transformation of a system from the analysis stage into the design stage -- which has also been viewed by many as a magical transformation.

The structured methodologies of the past suffered during this metamorphosis between analysis and design, I believe in part due to the lack of such a pattern language. I have watched several methodologists try to teach what they can do so easily, as they transition from analysis to design. To the uninitiated, it appears as if the methodologist is making it up on the spot. The reality is that the methodologist is using a great deal of acquired wisdom to aid in their transformation.

A pattern language approach provides an ideal solution to this transformation problem, as it is acquired wisdom that is contained within a pattern language.

Many of the popular object-oriented methodologies have skirted the transformation problem by introducing design issues into the analysis phase (e.g., identification of objects, class definition and the recording of inheritance). This has put many large projects at risk because of the complexities involved with making design decisions while one is not sure of what the system-to-be-built actually does.

To avoid the risk of failure and the cost of rework, a pattern language like Caterpillar's Fate needed to be developed as part of a comprehensive methodology. This methodology consists of an analysis phase and a design phase. The analysis phase is free of any object bias, leaving the questions of where or how to use objects as a design question. The design phase takes on different design strategies (e.g., object-oriented, object-based, structured, filter-tools, algorithmic, or various hybrids) depending on a number of technical or non-technical factors.

The object-free analysis phase consists of modeling activities that produce answers to the questions of:

| Analysis Question | Modeling Technique |
|---|---|
| What information is important in the problem domain? | Information Modeling |
| How does this information change over time and what incidents cause it to change? | Entity State Modeling |
| Who are the *users* and what *tasks* do they turn to the system to get accomplished? | User Task List |
| For each *user task* what is the data flow? | User-Task Partitioned Data Flow Diagrams |
| Exactly how does the human interface work? | Four Dimensional Human Interface Perspective |

This methodology is discussed further in [1], and [2].

**3.0 The Caterpillar's Fate Programming Language**

Caterpillar's Fate captures acquired wisdom, mine and my clients, around developing design solutions from fine analysis models. It documents what I do during the transition. It addresses the popular question, "How do I find the objects."

The format of Caterpillar's Fate is intentionally taken directly from Christopher Alexander's work[3] [4]. Two deviations have been made: I have included a section on "Suggestions" to the designer and I have left out a "Sketch" for each pattern.

The suggestions seemed too important to ignore just to emulate Alexander. It was page limitations that caused me to skip the sketches. Sketches do exist for each of these patterns, and I believe they are important.

So to begin with Caterpillar's Fate, I begin in the same manner as Alexander -- at the top with the first three design decision to be addressed:

---

When starting an architectural design from analysis documents, the issues of concurrency seem to be the ones to address first. The patterns addressing these issues are:

1. Concurrent Threads of Execution
2. Synchronization of Concurrent Threads
3. Collaborative Work Packets

---

*1. Concurrent Threads of Execution*

When a system contains processes that run either simultaneously or pseudo-simultaneously (a la operating system supported task switching), then careful planning is necessary.

The requirements documents often discuss function but rarely do the documents discuss exactly what functions will be available from which concurrent process. In fact such documents shouldn't mention how the functions are deployed, as it is a design decision. Therefore:

Identify the threads-of-execution that have the ability to exist independently from other threads that might exist in the system. In some cases, these threads will reside upon different machines, in other cases they will represent different *user*'s each with their own agenda working upon the same machine. In this pattern the word *user* means "an entity, external from the system, requiring service of the system." In some cases a user may be a human, in other cases it may be a device or simply the passing of time.

Suggestions - Name each thread, and write the purpose of each thread within a system of threads.

When the concurrent threads of execution have been identified, you can address the issues of *Synchronization of Concurrent Threads(2)*, and *Collaborative Work Packets(3)*. If your design has only one concurrent-thread-of-execution, then you are ready to address *Shape of Program(9)*.

*2. Synchronization of Concurrent Threads*

When the *Concurrent Threads of Execution* have been identified, you are ready to identify the synchronization that needs to occur between concurrent threads.

Often in the execution of a concurrent thread, there are situations where a thread needs to stop processing or remain in a certain form of processing until another concurrent thread arrives at some well understood state. Therefore:

Review each concurrent thread of execution, throughout the life-cycle of its execution and identify those points where a *signal* is either to be sent to or received from another concurrent thread. Name the signal for the situation it represents, identifying both the sender and the receiver. In this pattern a *signal* carries no information, it either has arrived or it has not arrived.

Suggestions - If your system seems to have a great deal of synchronization, it demonstrates that the various threads have a great deal of interdependency. A *great deal* is determined by noting the amount of concentration needed to understand the purpose and interaction of each signal as part of the whole system. If the level of concentration begins to exceed some reviewers patience, then the probability of creating defects increases as does the cost of enhancement. In such cases, return to the *Concurrent Threads of Execution(1)* pattern and change either the number of threads and/or their purpose, in such a way that design decisions from the *Synchronization of Concurrent Treads(2)* pattern become easy to understand.

Now with a clear understanding about how concurrent threads synchronize with each other, you are ready to look at the *Shape of Program(9)* and *Critical Region Protection(17)*.

*3. Collaborative Work Packets*

When you have a clear idea of what concurrent threads exist within your system, you are ready to look at the work packets that move between them.

Many systems contain the need for the processing of work to be split across processors, or defined and scheduled at one point in time to be accomplished at another point in time. An example of this is the transaction processing systems found in many business applications. In this pattern language, producers and consumers of work packets are the concurrent threads of execution. Over time, the producers and consumers may change. The nature of the work to be accomplished may change. As a result, it is important to design clear separation between the concurrent-threads-of-execution and the work passed between them. Therefore:

For each pair of concurrent threads, review the life-cycle of each thread and identify specific work to be initiated by one thread and continued by the second thread. Name the work packet for the information and work unit it manages or hides. For each work packet, identify the possible producer and consumer concurrent threads.

Suggestions: For each work packet you should be able to easily state what the work packet is responsible for. Usually the responsibility would read: "All that is known about a <fill in the blank>".

Once you have designed the work packets passed among the concurrent threads of execution, you are ready to design the *Work Packet Contents(4)*, and also start the design of the *Shape of Program(9)*.

---

Work packets, in some environments called "transactions" require some careful design. This design like 'concurrency' needs to be thought through early in the design as it represents a "big picture" perspective of how concurrent subsystems or concurrent-threads-of-execution will collaborate. Work packets have many details that need to be addressed early to avoid rework as the system design becomes apparent. The patterns in this section are:

4. Work Packet Contents
5. Work Packet Status Report
6. Work Packet Completion Report
7. Work Packet Priority
8. Work Packet Security

---

*4. Work Packet Contents*

Once a *Collaborative Work Packet(3)* has been discovered, named, responsibility recorded and each of the producing and consuming *Concurrent Threads of Execution(1)* have been identified, you are ready to design the internal contents of a work packet.

Work packets contain information and directions for a number of uses within a system. The internal design of a work packet is likely to become a mess if not conscientiously designed. Therefore:

For each *work packet*, develop a design which contains all the information necessary for the work to be accomplished. Such information includes: data, state, statement-of-need, and history. Data is used for specific processing. State contains information related to the present situation or condition that the work packet is in. A statement-of-need asks the consumer thread to help with some particular goal to be accomplished. History includes information related to the past experiences of the work packet, often including references to the concurrent threads that it has visited in the past (See *Data Knows Its Roots(21)*).

If *Work Packet Status Report(5)* or *Work Packet Completion Report(6)* are patterns of value to this design, then the work packet will also need to contain an identifier or name. As the patterns *Work Packet Priority(7)* and *Work Packet Security(8)* are addressed, they may have an impact on the contents as well.

To assure minimum impact to a work packet when either a producer or consumer changes, the work packet must never contain information related to its future -- i.e., it never tells the consumer how to go about processing. A consumer needs to be able to determine the kind of processing by querying about data, state, need and history. For example, an SQL phrase is a statement-of-need. The actual processing is left up to the consumer. On the other hand, directions to perform a particular function and then pass the work packet on to yet another consumer is an attempt at controlling the future.

Suggestions: Work packet design follows from the analysis documents. The data within a work packet can be determined by inspecting the information model developed in the analysis phase. State information can be deduced by looking at the entity state model. History is not likely to be found in the analysis documents, as it is dependent on the design decisions such as concurrent threads.

If a work packet begins to seem large, study the work packet for variation. If there are a number of kinds of a work packet, consider a design using sub-types of work packets if that would reduce the amount of data, state or history that needs to be included.

Where several different kinds of work packets are likely to travel to the same consumer for processing, where the processing is different depending upon the packet, develop a design where the differences in processing are contained within the work packet. A good test for such situations is to note if a work packet must answer questions about what type it is -- if so, consider a different design.

As work packets become defined, several other design decisions need to be addressed: *Work Packet Status Report(5)*, *Work Packet Completion Report(6)*, *Work Packet Priority(7)*, *Work Packet Security(8)*, and *Data Knows Its*

*Roots(21)*. With the identification of specific work packets along with their producers and consumers, you are ready to address the *Shape of Program(9)*.

*5. Work Packet Status Report*

When you have made your design decisions around *Collaborative Work Packets(3)* and *Work Packet Contents(4)*, you have enough knowledge to know if some report on the status of the work packet needs to be generated for the benefit of the producer.

In some designs, a producer concurrent-thread-of-execution generates work packets, ships them to the consumer thread and no longer has any interest in that work packet. For such situations this pattern has no value.

In other designs, the producer, often at the request of the user will want to monitor the status of the work packet. So the questions of what status information needs to be returned to the producer, what stimuli causes the report to be returned, what role does the work packet play and what role does the consumer concurrent-thread-of-execution play. These are the design decisions that need to be addressed at this point in the design. Therefore:

Assess the needs identified in your analysis and make the design decisions on the following questions:

1. When is the report sent? Here are some possibilities: only upon request by the producer;

   every time the status within the work packet changes;
   when processing within the consumer achieves a particular goal; or
   with the elapse of some unit of time.

2. What information should be in the report? Does the work packet require an identifier? Does the identifier need to be unique? Does a combination of pieces of information make a unique work packet identifier (such as user name and time sent)? What status information is part of the report? What data information? What history information? Is the report tailorable?

3. Is the desire for the report known at the time the work packet is passed to the consumer? If so, then design the work packet to be responsible for producing the report back to the producer, as it contains the knowledge of its producer.

   If the desire for the report occurs some time after transmission of the work packet, then choose a design where the consumer is responsible for producing the report. Such a report is not on one particular work packet but is a summary of all work packets within its control and knowledge.

4. Determine the needs for security around status reporting. Can a concurrent-thread-of-execution request or automatically receive a status report on work packets generated by some other concurrent-thread? Is this request available on a privileged basis? Is it password protected? Does the status report need to be encrypted?

Decide whether the work packet or the consumer has the prime responsibility to produce and send the report. Then determine what support is required from the other. Design a mechanism for communicating between concurrent-threads-of-execution to support your answers to these design questions.

Suggestions: Keep the work packet status report design as simple as possible! Do not put any extra "bells and whistles" in this part of your design. Your system needs to produce status reports, but its prime goal is not about producing these reports, it is about processing work packets in some way. Focus your design creativity there!

When these design decision have been made, you have a clear idea around the design of work packets. Assure that the specific *Shape of Program(9)* addresses the receipt of every status report identified.

*6. Work Packet Completion Report*

When you have made your design decisions around *Collaborative Work Packets(3)*, *Work Packet Contents(4)*, *Synchronization of Concurrent Threads(2)*, you are ready to make the design decisions around what happens when a work packet completes its processing within the consumer concurrent-thread-of-execution.

Upon completion of the processing of a work packet, there are many different possible design responses. These responses need to be considered now. Therefore:

Assess the needs identified in your analysis and make the design decisions on the following questions:

1. What gets returned? Is it a success/fail message? Is it an answer to the question contained within the work packet? Is it the work packet itself where data, state and/or history has changed?

2. If a failure report is returned, how much is communicated about the failure? In some situations, you will want to communicate:

   What the nature of the error was, and
   suggestions of what should be done differently.

   However, there are some systems that have a security requirement that will be breached if details of the nature of the failure are shared with the producer thread.

3. For purposes of security, does the response need to be encrypted? Does it need password protection?

4. Design the policy on how to deal with responses if the producer concurrent-thread-of-execution is not available at the point in time that the response is ready to be sent. This may occur in distributed systems because the network goes down, or the producing computer goes of-line, or for a number of other reasons. It is possible that the producer, requesting a response, will leave the system forever. How will this situation be handled?

Suggestions: The suggestions offered in *Work Packet Status Report(5)* apply here too. Keep the completion report simple and concise.

When these design decision have been made, you have a clear idea around the design of work packets. Assure that the specific *Shape of Program(9)* addresses the receipt of every completion report identified.

*7. Work Packet Priority*

Once you develop the design for *Collaborative Work Packets(3)* and while transforming the general *Shape of Program(9)* to a specific one, you need to make decisions about the order in which work packets are consumed.

In some systems, work packets are queued for processing, either at a later time, when a consumer's processing resource is available or cheaper, or because there are peak times when work packet arrival exceeds the processing capacity. When there is more than one work packet waiting for processing, there design decisions to be made around which work packet to select. Therefore:

Select a policy on how the next work packet is selected. In most analysis documents this is something that has been over looked. In such cases, "first-come-first-served" might be all that is needed. However, there may be requirements hidden in the form of general performance statements that may not be easy to find. Review the documents with the following options for selection in mind:

1. First-come-first-served.

2. Assigned priority. Work packet type may contain a priority or it might be a privilege priority associated with the producer that created the work packet.

3. Dead-line oriented. Each work packet has a dead line for completion and work is accepted in a manner that the fewest dead lines are missed.

4. Cost-penalty analysis. Each work packet has not only a dead line but also an associated cost for missing the dead line. Selection is based on minimizing the penalties.

These are basic design issues. It is possible that your system may need a combination of these approaches or may create a different selection policy. What is important that the policy for selection is consciously made, and designed into the consumer's specific *Shape of Program(9)*. Depending on the policy selected, *Work Packet Contents(4)* may need to be revisited to support the selection policy (i.e., it may contain a priority, or dead line).

Suggestions: Keep the selection system as simple as possible. Be prepared for any kind of change that may be suggested by: adding no "hooks" into the system now for possible future expansion and do not accept any assumptions or short cuts that will prevent a future growth.

"No hooks" because the future may not play out in the manner that you visualize it at this point in time. On the other hand, if it does come to pass, the hooks are not likely to be as well tested as today's working code, leading to suspicion plus any maintenance that may have occurred is not likely to have respected the unused hooks.

This "no-hooks" suggestion does not only apply to this pattern but throughout the design process. This was the first place the advice was likely to be important It will not be repeated throughout the this pattern language, though it is tempting.

When these design decision have been made, you have a clear idea around the design of work packets. Assure that the consumer's specific *Shape of Program(9)* addresses selection design decision and that the producer's *Shape of Program(9)* and/or *Work Packet Contents(4)* are refined as needed.

*8. Work Packet Security*

Once *Work Packet Contents(4)* and *Collaborative Work Packets(3)* have been addressed, you are ready to look at questions of security related to work packets.

When work packets can be constructed upon one machine and sent through a network to another machine, and the work packet contains sensitive information or work directives then there several security issues that need to be designed into work packets as well as the producer and consumer. Therefore:

Consider the following design issues:

- Does the information within the packet need to be encrypted. The answer is based upon the question of "would someone benefit from accessing the work packet being passed?", not "is it possible to access that information, given our hardware and configuration?". While you may believe that you have a secure network there is no assurance that is true now and in the future, as your configuration changes.

- In systems where some producers of work packets have privileges that others don't, either in information access or functionally permitted, then design the prevention of inappropriate function or information access into the producer, for human interface reasons and design the security checks to occur within the consumer. The work packet will need to carry the information necessary for the consumer to perform the security check.

- In some systems, aspects of security are performed by the regular review of the work packets consumed looking for anomalies. In other forms of security, an audit trail is used to understand what has happened. If either of these needs appear in the requirements, then develop a design for logging the necessary information and activity.

Suggestions: Security is usually left unaddressed or at best vaguely mentioned in most requirement and analysis documents. Most likely, you will find high security wishful thinking with little understanding of the depth of the issues or the cost. If this is the case and security is crucial, then stop all design activities, return to analysis and develop effective security models.

At this point, you have addressed a few security related issues, encryption, pass wording and logging are issues that have surfaced. As this pattern language matures, there are many more patterns that need to be discussed.

---

In this section we begin to look at the early design issues that occur around one program. This includes the early decisions you need to make in regards to the shape of the program, how stimuli will be acquired and responded to, and how a human interface is added to this program shape in such a way that the human interface may be replaced without impact to the rest of the system. This section also includes a pattern that revisits the concurrency questions to discover critical regions. The patterns in this section are:

9. Shape of Program
10. Systems Citizen Role
11. Decision Makers Role
12. Workers Role
13. Interface Role
14. Informational Role
15. Small Family Systems
16. Work Accomplished Through Dialogs
17. Critical Region Protection
18. Event Acquisition
19. Event Routing
20. Human Interface Role is a Special Interface Role

---

*9. Shape of Program*

Once you have a clear idea of the *Concurrent Threads of Execution(1)* within your system, the *Synchronization of Concurrent Threads(2)* across your system, and have itemized the *Collaborative Work Packets(3)*, you are ready to apply specific form to a general shape of a program. For systems with only one *Concurrent Thread of Execution(1)* this will be the first pattern considered.

At the early stages of transforming requirements documents into a software design, we have many issues to consider -- usually too many to keep in our heads at one time. As a result, it is easy to focus on optimizing design issues for the small part of the system that we have in focus at the moment to the peril of the "big picture" design. Therefore:

Accept a "typical big picture design" until it does not serve you (See *Small Family Systems(15)* and *Work Accomplished Through Dialogs(16)*). This pattern offers a typical shape of a program. While there are many possible shapes, this one seems to work well as a starting point on all the systems with which I have been involved.

The shape of this program is best described as tiered. On each tier, the objects have the ability to call for *service* upon design components on lower tiers or on the same tier but not on higher tiers.

The first tier contains objects whose responsibility it is to assure that this *Concurrent Thread of Execution(1)* is a good citizen within a community of concurrent threads each working towards their own agenda. See *Systems Citizen Role(10)*.

The second tier contains objects that are responsible for the decision making aspects of the system to be built. See *Decision Makers Role(11)*.

The third tier contains objects that perform work needed by the system. See *Workers Role(12)*.

The fourth tier contains objects that are responsible for hiding the interface to some external entity. See *Interface Role(13)*.

For this tiered system to work, there is a type of object that moves through the tiered systems, visiting the other objects. This moving object has the responsibility of providing the appropriate information as well as tailored functionality. Those familiar with object-oriented thinking will refer to these objects as polymorphic. See *Informational Role(14).*

The design decisions made during the use of this pattern transform the general shape of program to a specific one that supports the findings in the analysis documents. To accomplish this transformation, you will find it useful to consider the following patterns while accepting the tiered shape of programs: *Systems Citizen Role(10)*, *Decision Makers Role(11)*, *Workers Role(12)*, *Interface Role(13)*, and the *Informational Role(14).* When the tiered shape begins to fail, consider *Small Family Systems(15)*, and *Work Accomplished Through Dialogs(16).* As you begin to consider the issues around how mouse clicks, key strokes and the like are handled, you will find *Event Acquisition(18)* and *Event Routing(19)* useful.

Suggestions: Within a tier, services between objects might be called upon, but this should be kept to a minimum. The reason varies depending on the tier. Generally, as the interactions between tier peers grows, the reuse of either without the other decreases. This is particularly true for informational objects and worker objects.

Decision making objects contain knowledge of a particular application so they are not likely to be reused, so reuse is not an explanation. Interaction between decision makers requires maintenance engineers to master the operations of both decision makers before either one can be changed. Within limits this works, but taken to an extreme causes real problems. When confronted with such situations, revisit the analysis and design decisions that defined the boundaries and responsibilities of these two kinds of objects.

Four tiers may not seem sufficient for large systems. In some cases, I have seen a shape of program structure used recursively within one of the object roles described here, most often for decision maker objects and worker objects. I have also seen the internal working of some of these objects take on the form of a *Small Family System(15).*

When the general shape of program has been transformed into a specific shape for your system, assure that every signal identified in *Synchronization of Concurrent Threads(2)* and every work packet identified in *Collaborative Work Packets(3)* is accounted for.

With the general tier shape transformed to a specific one reflecting the findings of the analysis documents, you are ready to refine the system for various measures of goodness, performance, and other factors beyond the scope of this early architectural pattern language.

*10. Systems Citizen Role.*

When there is a clear idea of a *Concurrent Thread of Execution(1)* and the *Shape of Program(9)* begins to take shape from the general to the specific, you are ready to look creating a single systems citizen object.

Most programs reside on platforms where there are likely to be several applications running at the same time (e.g., Mac OS 7.X, OS/2, Windows 3.X, etc.). In these environments there are expectations that the operating environment has for each application running. Each of these environments have their own way of operating. There are likely business reasons to design an application that can be moved from platform to platform with ease. Therefore:

Design a single object that is responsible for knowing the protocol that applications must follow to be "good citizens" within a community of applications. As you consider how the particular operating environment handles issues around mouse clicks, key strokes and the like are handled, you will find *Event Acquisition(18)* and *Event Routing(19)* useful.

Suggestions: The information about the responsibilities of a systems citizen are not found in the analysis documents. They are usually found in a programmer's reference manual or tutorial manual for the operating environment chosen.

With this pattern mastered, you are in a good position to understand how *Decision Makers Role(11)* and *Human Interface Role is a Special kind of Interface Role(20).*

*11. Decision Makers Role.*

When the *Shape of Program(9)* begins to take shape from the general to the specific, and the *Systems Citizen Role(10)* becomes defined, you are ready to design the high level policy making aspects of your application.

Most "real life" systems have some interesting design issues around controlling the overall operation of an application. These issues include the fundamental behavior of your application -- the work-flow, the control, the highest level decision making about the behavior of your application. Often it is these decisions that your customers will use to differentiate your application from your competitors.

If these controlling issues are not respected as interesting in their own right, then the policy/decision making activities are likely to be distributed across a number of objects. This creates significant difficulties for the maintenance activities. Distributed policy/decision making requires a maintenance engineer to master the workings of a large number of objects before any changes to policy/decision making can occur. Therefore:

Create an object or a number of objects whose prime role is to be responsible for the decision making activities of your application.

Suggestions: Decision making objects look funny. They often have only one external method -- 'doIt'. Nevertheless they are key to building systems that are maintainable and reusable. The reusable component comes from placing all that is not likely to be reused (i.e., the policy/decision making aspects of this application) in one or a number of objects that are not likely to be reused. Decision making objects are not reused because they are application specific, unless the application reused involves porting the same application to another environment.

Input for the policy/decision making object is usually found in the process specification and is what remains when all the lower level operations can be allocated to worker objects(12), informational objects(14) and interface objects(13).

At this point you have a clear idea of how your application will control the work to be accomplished. You are ready to connect the *Workers Role(12)* objects, the *Interface Role(13)* objects, and the *Human Interface Role is a Special Interface Role(20)* object. It is quite likely that the decision maker will handle a number of *Informational Role* objects, as they are passed among the worker objects and interface objects.

*12. Workers Role.*

When the *Shape of Program(9)* begins to take shape from the general to the specific, you are ready to design the objects that do work towards helping the application move closer to its goal.

Within many programs there are design decisions that can be combined together to provide a number of related services that help an application move closer to its goal. Therefore:

Build specific objects with the responsibility to do work that the decision makers would find useful in exercising the policy/decision making responsibilities. Collections are a common kind of worker object.

Suggestions: The number one goal of a worker object is to off-load any work from decision makers that is not directly involved with controlling the application. A secondary goal is to build worker objects that are likely to be reused in similar but different applications.

Worker objects are usually easy to find from an Event-Partitioned Data Flow Diagram. There are often data stores, or are hidden within data stores. The process specifications are used to confirm that they are worker objects and to identify the methods.

When you have identified the worker objects, you are ready to look to the internal design of these objects, comfortable that the big picture will fit with the small picture that now becomes of interest to the designers.

*13. Interface Role.*

When the *Shape of Program(9)* begins to take shape from the general to the specific, you are ready to design the objects that hide the specific behavior of entities that are not part of your system.

Systems are not built in isolation, they interface with something else -- it might be hardware devices, or software not under the control of the designers. Regardless, since it is not under the control of this system design, these external entities may change their behavior. Therefore:

To protect your system-to-be-built, hide the external entity's behavior within an object whose responsibility it is to provide high level abstract services to the rest of your system.

Suggestions: Interface objects can be found from the Event-Partitioned Data Flow Diagrams; they are usually the external entities. The abstract services can be determined by looking at the process specifications.

When you have identified the interface objects, you are ready to look to the internal design of these objects, comfortable that the big picture will fit with the small picture that now becomes of interest to the designers. There is a special kind of interface object that hides the human interface from the decision maker objects. See *Human Interface Role is a Special Interface Role(20).*

*14. Informational Role.*

When the *Shape of Program(9)* begins to take shape from the general to the specific, you are ready to design the objects that move through the system, visiting the interface objects, the worker objects at the direction of the decision making objects.

We have seen a great deal of value from the use of polymorphism. The *Shape of Program(9)* structures a number of objects into a rigid tiered system, similar to the old structured design style of thinking. There were some advantages to such a structure, but polymorphism was not possible. The closest the structured methods could come is to acknowledge the value was the passing of complex data structures known as 'tramp data'. This was considered not the best of designs, as the knowledge of the structure of the data was spread across an application. Object-oriented thinking gave us a way to solve this problem by passing objects. Therefore:

Create objects whose responsibility is to deliver specific information at the appropriate places within an application, without letting the structure of the information become known. These are known as *informational objects*.

Suggestions: Where ever you discover the design needs to ask an informational object what its type is to resolve a number of IF statements, SWITCH statement, etc., then stop and look for a way to put the type related work within the informational object.

Informational objects are found by noting how information flows on an Event-Partitioned Data Flow Diagram. Solid meaningful pieces of information that flow from or to *Interface Role(13)* objects and that have some interesting processing associated noted in the process specifications suggest informational objects. This is especially true if you discover the data dictionary suggests that there are a number of types or variations on this information.

Smalltalk experts will recognize the similarities between *informational objects* and the *model* component of the *model-view-controller* triad. This is discussed further in *Human Interface Role is a Special Interface Role(20).*

When you have identified the informational objects, you are ready to look to the internal design of these objects, comfortable that the big picture will fit with the small picture that now becomes of interest to the designers.

*15. Small Family Systems*

When the *Shape of Program(9)* begins to take shape from the general to the specific, you will encounter times when a tiered system creates awkward designs. In such cases, you do not discard *Shape of Program(9)* but just augment it with another 'mini-shape' for the portions of the system that you are considering. Often the internal design that comes from *Human Interface Role is a Special Interface Role(20)* is benefited by this pattern.

In particular, you will find that there are design segments that are well served by building a small number of objects that work well together as a team. The key feature that makes this design segment unique is that in this 'small family' of objects, no one object takes on the control responsibilities for the others. Therefore:

Design the small family, as a family but make sure it is a 'healthy family'. In a healthy family, each object has a distinct responsibility, easily differentiated from any of the other family members. For each service provided by this family to the system, there is one object that takes the control responsibility and calls on the other objects for help in accomplishing this goal. The classic Model-View-Controller is an example of a small family system.

Assure that the boundaries of responsibilities among a family are never violated. When a boundary of responsibility is violated, it creates a situation where it is not clear which object is responsible for accomplishing a given activity. In such cases, a maintenance effort is likely to get it wrong, causing some activity to be performed twice or causing an activity to not be performed at all.

Depending on your language and operating environment, you may not be able to implement it in this manner, from a design perspective, think of the small family of objects as a single object when placed within the tiered structure of *Shape of Program(9)*.

Suggestions: Small family systems are composed of a few kinds of objects -- two, three or rarely four kinds objects -- is a good guideline. I have never seen more than four kinds of objects work well as a small family.

Small families are usually not identifiable from the analysis documents, they are usually created to solve a situation that either supports simultaneously a number of design goals, or to simplify an awkward structure developing in the tiered structure. They are clever inventions often adapted from one's previous experiences.

Small family systems were chosen as a secondary design strategy after tiered systems because the amount of effort a maintenance engineer needs to apply to master how the family works. Use this 'effort to master' measure to guide your design decisions and documentation plans.

When you have identified the need for a small family system, you may be ready to look to the internal design of these objects, comfortable that the big picture will fit with the small picture that now becomes of interest to the designers.

*16. Work Accomplished Through Dialogs*

As the *Shape of Program(9)* evolves from the general to the specific, you will encounter times when two objects need to invoke methods in each other to get their job done. In such cases, you do not discard the *Shape of Program(9)* tier structure but just augment it with a design decision to allow some objects to call methods in each other.

There are situations where the required work to be accomplished by the system involves the blending of two objects responsibilities. Neither object contains all of the information or processing ability to achieve the requirements. Combining the two objects together does not seem to be a wise idea, each of the responsibilities seems to be well formulated, and there does not seem to be an undiscovered decision maker object to control the interactions between these objects. Therefore:

Choose a design where the two objects carry on a dialog with each other. The first object invokes a method in the second object that in processing, calls a method in the first. It is this second object that initiates a *dialog* with the first object, otherwise we just have a tier connection.

Suggestions: A dialog between two objects is easily understood in moderation.

Between two objects, if the method that is invoked on the dialog (i.e., when the second object calls the first), is one of a *questioning nature* then it will be easily understood. Questioning nature is a kind of method whose sole purpose is to provide information, or answers that help the second object further its work.

The counter to questioning nature is *controlling nature*. A method with controlling nature changes the state of the first object and in the worst case, causes the first object to interact with still more objects in a controlling manner. The deep implications of a controlling nature dialog is hard to comprehend. Reviewers and maintenance engineers are likely to miss subtle implications. Avoid controlling nature dialogs!

A dialog that involves more than two objects is called a *chain-letter dialog* and is to be avoided at all costs. Picture a design where the first object sends a message to a second object. The second object calls a method in the third and eventually the Nth object asks the first object for some information. In such a case, there is a dependency involving

all N objects. No maintenance change can be made to any one object until all N objects have been considered. This violates all that we have learned about information hiding and separation of concerns. Never carry on a chain-letter dialog.

With the judicious use of questioning nature dialogs to refine the *Shape of Program(9)* design, you can return to the transformation of the shape from general to specific.

*17. Critical Region Protection.*

When *Concurrent Threads of Execution(1)* have been identified and for each of those threads, the *Shape of Program(9)* has moved from general to specific, you are ready to look for the potential critical region problems.

Concurrent systems often use the same resources, usually in the form of interface objects(13), informational objects(14) or worker objects(15). Occasionally they will share decision making objects(11). If two concurrent-threads-of-execution access the same object at the same point in time there is the possibility that they will cause harm. This harm might effect the shared object, the work being performed for the other concurrent-thread, or in rare cases the invoking concurrent-thread's own work. Therefore:

For each concurrent thread, review the specific shape of program and identify the shared objects. For each shared object determine if protection from simultaneous access is needed. If so, develop a protection mechanism that assures safe use.

Suggestions: There are several design solutions to this problem and they can be reviewed in any operating systems text. Some solutions include: blocking the second thread upon entry to the critical region; requiring each thread to get permission before entering the critical region; aborting a low priority thread's work when a high priority thread enters the critical region; or removing the critical region by giving each thread its own data memory space.

When this pattern has been addressed, you can be sure that the most difficult defects to discover have been prevented.

*18. Event Acquisition*

When the *Shape of Program(9)* begins to take shape from the general to the specific, and the *Systems Citizen Role(10)* is being considered, this pattern will help resolve some design options.

Every platform deals with *events* in their own unique way. An event being the signal that indicates a mouse click, a keystroke or the like, has occurred. Some operating environments provide a great deal of support, where others leave the responsibility more heavily to the system designer. Nevertheless, the shape of program needs to be tailored to accommodate a thought through, consistent design. Therefore:

Review the event acquisition capabilities provided to designers. Choose capabilities that makes sense for your application and refine the shape of program. Key design questions to consider include:

- Exactly how is an event acquired? Does the operating environment notice the event, acquire it and decode it? Or is some or all of that activity left for your design? Are any events treated as unique or handled in a different manner? Are there options where event acquisition provided by the operating environment can be over-ridden? When would you want to do this?

- What operating environment capabilities do you want to accept? How does that impact the shape of program(9)? Are these capabilities able to be contained within the systems citizen object(10)? If not, what consequences does that imply for movement of the application to a different platform?

- When do you provide your own event acquisition design? How does that impact the shape of program(9)? An example might be when the silhouette of a graphic is being dragged across a canvas. For what reasons will you accept not using the platform's supported event acquisition mechanism (e.g., performance, or function not supported).

- Are hardware interrupts supported by the operating environment? Is there a particular design that the operating environment expects from your 'hardware driver'? How does shape of program need to be modified to support these design decisions?

- Does the operating environment support the notion of an 'event consuming object' *registering* for particular events? If so, then how does shape of program change to take advantage of such a powerful capability? Document when will you not use the registration capability?

Suggestions: If this is the first time you have designed an application for this particular operating environment then sketch a number of variations on shape of program, have them reviewed by a team and build a number of prototypes. Evaluate each prototype for its advantages and disadvantages. Document your findings for others to follow and for you to review as you work beyond your learning curve.

When you have addressed this pattern, you will have refined *Shape of Program(9)* to better meet your design needs. Return to *Shape of Program(9)* and continue to transform the general shape to a specific one. In the future, patterns that address how systems can be tailored during initialization will be referenced from here.

*19. Event Routing*

When the *Shape of Program(9)* begins to take shape from the general to the specific, and the *Systems Citizen Role(10)* is being considered, this pattern along with *Event Acquisition(18)* will help resolve some design options.

Every platform deals with *events* in their own unique way. an event being the signal that a mouse click, a keystroke or the like, has occurred. When these events occur, there are a number of different approaches to assuring that the event is received and acted upon by all objects that should respond. Therefore:

Refine the shape of program to show how events will be consumed. You have several design decisions to consider:

- Is each event to be routed to only one object or are there likely to be a number objects that want to be informed when the same event occurs?

- How is the routing handled? Is it hardwired as we might find in an interrupt handler? Is there a negotiation activity where event consuming objects (usually decision makers or human interface objects) will answer if they recognize/are interested in a particular event when it arrives? Is registration the right design for your application? (Registration is a design idea where objects register for events with a dispatcher. When the event arrives, the dispatcher routes it to the object that registered for it.)

- Under what situations will you consider circumventing the operating environment's way of routing events?

Suggestions: Keep event routing as simple as possible. Dynamic registration of events, that is, registration that occurs continuously throughout the life of a system, yields a system that is impossible to understand by reading the code. A maintenance engineer will have to watch how events are continually registered and removed from a debugger. In such situations you never know if you have seen all the possible combinations.

When you have addressed this pattern, you will have refined *Shape of Program(9)* to better meet your design needs. Return to *Shape of Program(9)* and continue to transform the general shape to a specific one. In the future, patterns that address how systems can be tailored during initialization will be referenced from here.

*20. Human Interface Role is a Special Interface Role*

As you transition *Shape of Program(9)* from a general form to a specific design, you will be putting care into how the decision maker objects(11) get direction from humans. At that point, this pattern should be considered.

The human interface is key to providing information and direction to an application. It is found in numerous places throughout an application. Another factor to consider is how difficult it is to port a system from one platform to

another if the human interface has been allowed to pervade the system in an unplanned fashion. As a result, the shape of program needs to be refined to include a human interface component that is isolated from significant portions of the system-to-be-built. Therefore:

Create a special kind of interface object(13) that hides the human interface specific design decisions. The decision maker(11) objects are the most likely to need input from the human. Refine the shape of program to include human interface objects accessed by the decision maker objects. The human interface objects deals with the human interface/platform specific issues and leaves the more abstract decision making and control of processing up to the decision maker object.

Worker objects and interface objects may also need the same kind of special human interface object to support their lower level work. This would appear in the form of dialog boxes and the like. Informational objects may also carry around a number of special human interface objects, each providing a unique view on the data in the informational object.

Suggestions: The behavior of human interface objects can be found in the details of the Four Dimensional Human Interface Perspective model. Deciding what is the responsibility of the human interface role object and what is the responsibility of the decision maker can be tricky. Develop several alternative designs and discuss them with a peer. Select between the designs by using the guiding principle "simplicity in explaining to someone else."

This pattern has helped you add human interface capabilities to you shape of program. You can return to *Shape of Program(9)* to further refine the general shape into a specific one.

You can also look at the internal design of the human interface object. You will discover that the *Small Family Systems(15)* pattern is quite applicable for the internal design. You will also discover that *model-view-controller* is a perfect small family system, and that the decision maker object will be passing the *model*, as an informational object(14), into this human interface object. The actual behavior of the controller and view will be hidden within this human interface object. You will be able follow the design decisions made about *Event Acquisition(18)* and *Event Routing(19)*.

---

In this section we look to the patterns intended to handle the design of data. As this is a pattern language that is best described as "work in progress", this section has only one pattern at this moment. It needs to be expanded, but I did not want to leave this one pattern out given its importance. The pattern is:

21. Data Knows Its Roots

---

*21. Data Knows Its Roots*

When you have identified *Collaborative Work Packets*, *Informational Role* oriented objects, or any other "entity" that contains information, you are ready to determine if "roots need to be known."

Good systems live for quite some time and over time change. Upgrades, added functionality, and addressing new markets are some of the reasons systems change. As new versions are released, it is crucial that "work-in-progress," no matter in what state, be as accommodated by the new version as the older version. At the same time, with expanded functionality, work packets, informational objects and the like must be free to improve as new versions are designed. Therefore:

For any "entity" that contains information (work packets, informational objects, etc.) and either

- has a possibility of persisting beyond the life time of the program that created it, or
- may move between concurrent-threads-of-execution, then

design the information to know the following attributes:

- record the type and a version identification for this informational entity,
- record the version of the program that created it, also include the date of creation and any information on the environment that can be accessed (machine type, machine configuration, OS version, file system version, etc.).
- record the date-of-last-access and date-of-last-change,
- for every concurrent-thread-of-execution visited, record the thread's version and date-last-visited,
- for every application that accesses this information, record the application's version and date-last-visited, and
- embed within this information knowledge about its stored data structure format.

Suggestions: For large entities, this additional "roots" information adds little to the overhead in terms of time and space. For very small entities, the overhead may be significant, but in low quantities it still remains valuable. For high volume, small entities, get creative before you discard the design idea. Consider the design of a transaction log which will record this information and during slack times, works at condensing the information at the cost of processing time.

Upon completion of this pattern you have assured that your persistent objects will have the information necessary for future versions of your system to grow and at the same time assure work in progress at any stage can be accommodated.

This is the end of the Caterpillar's Fate programming language. In the next sections, the use of this pattern language is discussed.

## 4.0 Use of Caterpillar's Fate

A reader might assume that one can traverse through Caterpillar's Fate without any kinds of iteration or rework. This is not the case. In some cases, the transition from analysis into design points out significant wholes or conflicts in the analysis documents. In other cases, as the design begins to emerge, it becomes clear that certain design issues were not considered. Users of Caterpillar's Fate do experience iterations of many sorts. That is not necessarily a flaw in the methodology, the pattern language or the project. Building large systems is an activity of learning what the system does. Learning often occurs through iteration. What is a flaw is to know that you need to iterate and to not re-analyze or re-design.

## 5.0 Status of Caterpillar's Fate

This document contains about one third of all the patterns that I have become aware I use. The rest of the patterns were left out for page limitation reasons. I use the phrase "have become aware" because I discover more common design wisdom every time my clients and I use Caterpillar's Fate. Furthermore, there are occasions where insight through use causes me to rework the patterns I have already documented, thus Caterpillar's Fate is to be seen as work-in-progress.

Nevertheless, this work has a high degree of maturity. Caterpillar's Fate has been used by my clients, since 1992, to construct several real-life systems: an investment banking system, an interactive voice-response system, a hand-held computer application and the control of electricity. Teams ranged from three to twenty people. In all cases, the pattern language was communicated within a methodology course followed by personal coaching as necessary. This paper is my first attempt to communicate these concepts solely in a written form.

## 6.0 Issues

As I developed the text for this pattern language, I was shocked at how rapidly I consider these patterns while transforming a system analysis into a first design. This leads to a concern of mine about the amount of effort that another person might have to put into mastering this language before he/she can use the wisdom that I have acquired. This is part of the new experiment exploring *pattern languages* for developing programs. Can pattern languages be

used by someone other than the author, and without the aid of a masterful teacher? For us to know if that is possible, written pattern languages need to exist.

This is why I have written this paper. It serves as the test-bed to see if pattern languages such as Caterpillar's Fate can be distributed widely.

A second concern I have with Caterpillar's Fate is the dependence on objects as the design paradigm. Alexander's pattern language is implementation material free. Following his lead, I would like to see the transformation from an object-free analysis be able to metamorphosis into any of a number of design strategies. Some of the patterns in Caterpillar's Fate were presented in an non object-bias manner, such as *Concurrent Threads of Execution(1)*, but others, such as *Work Accomplished through Dialogs(16)*, have a most definite object-bias. The bias free presentation of this acquired wisdom remains a puzzle.

Another issue to be resolved, not only for Caterpillar's Fate but for all pattern language authors, is how to incorporate other pattern language author's work, and honor their work. There is at least one other pattern language that I'd like to disassemble and incorporate parts of it within Caterpillar's Fate. Can I do this and still respect the original author's work? I may want to rewrite every part except the idea, and I may not even present the author's idea as he/she originally attended. As a community we need to discuss this and related issues.

A final issue has to do with the fear of public attack. Caterpillar's Fate describes details of how I generally design programs. It is risky to say, "this is what I do." I am certain there are programs if built following the wisdom of Caterpillar's Fate, would yield something that I would be embarrassed to show my colleagues. I claim the right to not follow any of my wisdom when I believe that it will not serve me well.

Also this is how I design today; I claim the right to learn new and different ways to design and as a result to create a new version of Caterpillar's Fate.

## 7.0 References


[1] Kerth, N., "A Structured Approach to Object-Oriented Design," Addendum to the OOPSLA Proceedings 1991.

[2] Kerth, N., Rhodes, R. & Burley, J., "How to Deliver 20,000 Lines of Code with only Four Defects for under $2.00 Per Line of Code," Pacific Northwest Software Quality Conference, Fall, 1990; Invited Paper.

[3] Alexander, C, "The Timeless Way of Building," Oxford University Press, 1979.

[4] Alexander, C., et al., "A Pattern Language," Oxford University Press, 1977.