

Design Patterns for Graceful Degradation *

Titos Saridakis

NOKIA Research Center
PO Box 407, FIN-00045, Finland
`titos.saridakis@nokia.com`

Abstract

The term *graceful degradation* describes the smooth change to a lower state of some system aspect as a response to the occurrence of an event that prohibits the manifestation of the fully fledged system behavior. Graceful degradation has appeared in a variety of domains, from image processing and telecommunications to shared memory multiprocessors and multi-modular memory systems. In each domain, graceful degradation has targeted a different system aspect, e.g. image quality, voice quality, computational capacity, memory access throughput, etc. However, irrespectively of the system aspect that has been gracefully degraded, the basic concepts behind the corresponding mechanisms have been similar in all the domains where graceful degradation has been used. This paper presents two design patterns that capture design principles of graceful degradation. The first of the presented patterns captures the general idea behind lowering the state of a system aspect in the occurrence of an unsolicited event that affects the quality of that aspect. The second pattern elaborates on a method for lowering gracefully the system state.

1 Introduction

All systems, including software ones, may experience during their execution a number of unsolicited events that cause them to fail in delivering the exact functionality and preserving the exact system qualities that they would have in the absence of those events. Such unsolicited events include internal errors like the failure of hardware or software constituents of the system, and exceptional conditions in their operational environment like incorrect user operations, sudden temperature jumps and low energy levels. In the context of this paper we use the term “*error*” to refer to such unsolicited events.

Poor system design that does not consider the occurrence of such events may result in arbitrary failures of the system, while benign design that considers the possibility of errors may allow the system to fail safely by halting. Although there is a big difference between shutting down safely and letting the system continue operating arbitrarily and being a hazard to its environment, these two cases have a common, undesirable characteristic: the system is not useful to its user anymore.

On the other hand, fault tolerant system design introduces countermeasures for errors based on techniques such as rollback recovery and active replication, which allow the system

*Copyright 2005 © by NOKIA. All rights reserved. Permission is granted to copy for EuroPLoP 2005.

to continue operating correctly despite errors and exceptional operational conditions. However, as the size and the complexity of a system increases, and as the environments in which the system must operate start to vary, it becomes impractical to add fault tolerance mechanisms for all possible errors that may occur in all possible environments. Non-life-critical systems cannot always afford rendering fault tolerant all their constituents because of the increase in the system complexity, or the limits imposed by their embedded nature, or even the increase in their production costs.

There is an increasingly large number of systems that cannot afford fault tolerance mechanisms for dealing with various kinds of errors, yet they require a less radical reaction to errors than halting safely. Such systems can respond to an error by removing from the service it delivers the part of the functionality that has been affected by the occurred event. To render more concrete the lowering of the system functionality when errors occur, let's consider the example of a window manager, which is responsible for drawing on a computer screen the windows that contain application GUIs, command shells, etc. Let's assume also that the window manager is composed from a set of software components, each responsible for some functionality like 3D shadow visual effects, cursor animation, smooth dragging of one window over another, a frame for each window, and a menu-bar on the top-part of the window frame.

An error in the software may cause the 3D shadowing component to fail, hence the window manager is no longer capable of calculating and drawing the shadow of windows on the screen. In the lack of a fault tolerant mechanism that could fix the damage caused by this error (e.g. instantiate a new component that calculates and draws 3D shadows for the windows), the window manager is left unable to deliver the 3D shadow effect to the windows on the screen. Rather than halting, the window manager can continue delivering the rest of its designated functionality except from the 3D shadows.

In fact, the system may respond in the same way to errors on other components of the system like the one responsible for the cursor animation, or the smooth dragging, or the window frame, etc. Rather than halting its operation, the window manager will continue executing without any cursor animation, smooth dragging effect or frame around the windows on the screen. As long as the window manager is able to convey the user input at each window to the corresponding application and the output of that application back to the corresponding window, the system may continue its execution, although in a less appealing state than the full-fledged state in which it had started its execution.

This gradual lowering of the system functionality as a response to errors is called *graceful degradation*, and it has been used in a number of domains to describe the smooth changing to a lower state of some system aspect. The system aspect that degrades to a lower state ranges from the content quality in image processing [4, 7, 9] and in telecommunications [26], to the system performance in shared memory multiprocessors [15, 24], multi-modular memory systems [5, 6, 11] and RAID systems [3], and to the delivered system functionality in distributed agreement protocols [12], systems specified using formal methods [8, 25] and component-based software [20, 22]. Table 1 contains a summary of the different domains where graceful degradation has been applied.

This paper presents two design patterns that capture the design principles of graceful degradation. The first pattern captures the general idea behind lowering the state of a system aspect in the occurrence of an error that affects the quality of that aspect. The second pattern elaborates on a method for lowering gracefully the system state.

DOMAIN	INTERPRETATION
Image processing	Rather than discarding a compressed image that cannot be reconstructed to its original resolution, the <i>image quality</i> can be gracefully degraded to lower levels [4, 7, 9].
Telecommunications	Rather than closing a connection that is dropping some packets of voice data, the <i>voice quality</i> delivered to the end-points is gracefully degraded to lower levels [26].
Shared memory multiprocessors	Rather than stopping a computation on a multiprocessor system when one of the processors fails, the tasks assigned to the failed processor can be rescheduled on the remaining processors resulting in the graceful degradation of the <i>computational power</i> of the system [15, 24].
Multi-modular memory	Rather than rendering inaccessible the memory when a one of its modules fails, the data of the failed module are recovered from stable storage and distributed among the remaining modules resulting in a graceful degradation of the <i>memory access throughput</i> of the system [5, 6, 11].
RAID	Rather than rendering the RAID (Redundant Arrays of Inexpensive Disks) system inaccessible when one of the disks fails, the data of the failed disk are reconstructed from the error correction codes on the remaining disks and distributed among those resulting in the graceful degradation of the <i>disk access throughput</i> of the RAID system [3].
Distributed agreement	Rather than failing to reach an agreement when one or more members of the distributed agreement group fail, the group can reach an agreement on a value less close to the expected one resulting in a graceful degradation of the <i>computational accuracy</i> of the agreed value [12].
ABFT	Rather than dropping the degree of fault tolerance that an ABFT (Algorithm-Based Fault Tolerance) system provides when one of the processors fails, the remaining processors are reorganized in a smaller number of fault tolerant groups with the same degree of fault tolerance resulting in the graceful degradation of the <i>computational throughput</i> of the ABFT system [25].
Formal specification	Rather than causing a system failure when a set of constraints that enable some system functionality is not met, an alternative, weaker set of constraints that still hold can enable an alternative system functionality resulting in the graceful degradation of the <i>constraint strictness</i> satisfied by the system during its execution [8].
Component-based software	Rather than letting the whole system to fail when an error occurs on a component of the system, the error affects only those components in the system that directly or indirectly depend the failed component. This results in the graceful degradation of the <i>system functionality</i> to a subset of it that has not been affected by the error [20, 21, 22, 23].

Table 1: Summary of domains where graceful degradation has been applied.

2 State Decrement

The **State Decrement** pattern captures the basic idea behind graceful degradation without entering into the details of how the lower state is defined.

2.1 Example

A window manager WM draws on a computer screen the graphical containers of applications that run on the computer. In addition to the basic window management functionality (i.e. creating, moving, re-sizing, minimizing, restoring, maximizing, and destroying windows) WM also provides graphical functionality such as 3D shadows around the windows, cursor animation, frames that surround windows, and a menu-bar at the top-part of the window frame. While the system is running, a memory leak causes an error when the 3D shadow function is called to calculate the shadow of a re-sized window. The fault tolerance mechanism integrated with WM does not deal with such errors. It is expected that WM does not crash or halt its operation despite the error that has occurred. Rather, it is desirable that WM continues executing and delivering its basic window management functionality plus the rest of its designated functionality except the 3D shadows (i.e. cursor animation, window frames and menu-bars).

2.2 Context

The context, in which the **State Decrement** pattern can be applied to gracefully degrade a system aspect, is defined in terms of the following invariants:

- The system aspect (e.g. functionality, performance, content quality, computational accuracy, etc) under study is modular, i.e. there is more than one state in which the given system aspect may be present in an execution of the system.
- The system can continue executing while the given aspect switches from one state to another.
- The impact of an error on the given system aspect is well-defined, i.e. the states of the system aspect that are unfeasible after the occurrence of an error are known to the system.

The first context invariant ensures that the system aspect under study consists of a set of modules and not all modules are absolutely essential for the given aspect to be part of a system execution. For example, a component-based system delivers its functionality by combining the functionality provided by its constituent components, and the system is still able to deliver part of its designated functionality although not all constituent components are present in its composition. The second context invariant ensures that the system does not halt its execution when the state of some aspect is lowered. Halting the system and restarting it in a different system-wide state (including a system-wide state that contains a substate of lower quality for the given aspect) falls outside the scope of this pattern. Finally, the third context invariant ensures that there is a known association between an error with the modules of the system aspect that it affects (e.g. in a component-based system, a detected error is located on one component or one group of components).

2.3 Problem

In the above context, a problem that arises is the following:

How can the system survive an error that causes a failure in some part of a system?

The **State Decrement** pattern solves this problem by balancing the following forces:

- Removing completely from the system execution an aspect that has been affected by the occurrence of an error has a high cost for the system as well as for its environment (e.g. user of the system). Moreover, in certain cases it is not feasible to continue the system execution once an aspect is completely removed.
- Allowing the system to continue executing unchanged although some aspect is affected by an error may cause the failure of the entire system; hence, it is unacceptable.
- Fault tolerance mechanisms that may be in place inside the system can deal with some, but not with all, errors that may occur during an execution of the system. In addition, other types of unsolicited events besides errors (e.g. alarms related to conditions of the operational temperature, resource saturation, and power energy levels) are not covered by the fault tolerance mechanisms.

2.4 Solution

Devise a function that takes arguments an event, a system state (the current one), and a set of constraints on the system state (those that the new system state must respect). This function calculates the impact of the event (e.g. the error) on the given state and produces a substate of the state given as a parameter, which conforms to the specified constraints and is free from the caused error.

N.B.: The solution suggested by the **State Decrement** pattern to the problem of surviving errors does not address issues related to the detection of such event. For cases such as the energy power dropping below a certain threshold and the environment temperature or the resource saturation exceeding an acceptable level, alarm mechanisms are expected to be in place to report these events. Another type of execution errors (e.g. timing and value errors on some system functionality), for which appropriate error detection capabilities are required to observe their occurrence and notify the rest of system about it. Design patterns that address the error detection and notification problem are studied elsewhere [14, 16].

2.5 Structure

The solution to the problem of surviving the occurrence of errors that is described by the **State Decrement** pattern outlines the following entities:

- The system *aspect*, which is the part of the system that is affected by the occurrence of an error. The *aspect* has more than one state in which it can be during a legitimate system execution.

- The *assessor*, which is the part of the system that calculates the impact of an event on the *aspect*. The *assessor* is responsible for deciding what is the new, error-free state that the given aspect must load after the occurrence of an error.
- The *loader*, which is responsible for loading the new state decided by the *assessor* to the aspect of the system that suffered from the occurrence of the error.
- The event *notifier*, which has the responsibility of (detecting and) notifying the *assessor* about the occurrence of an event.

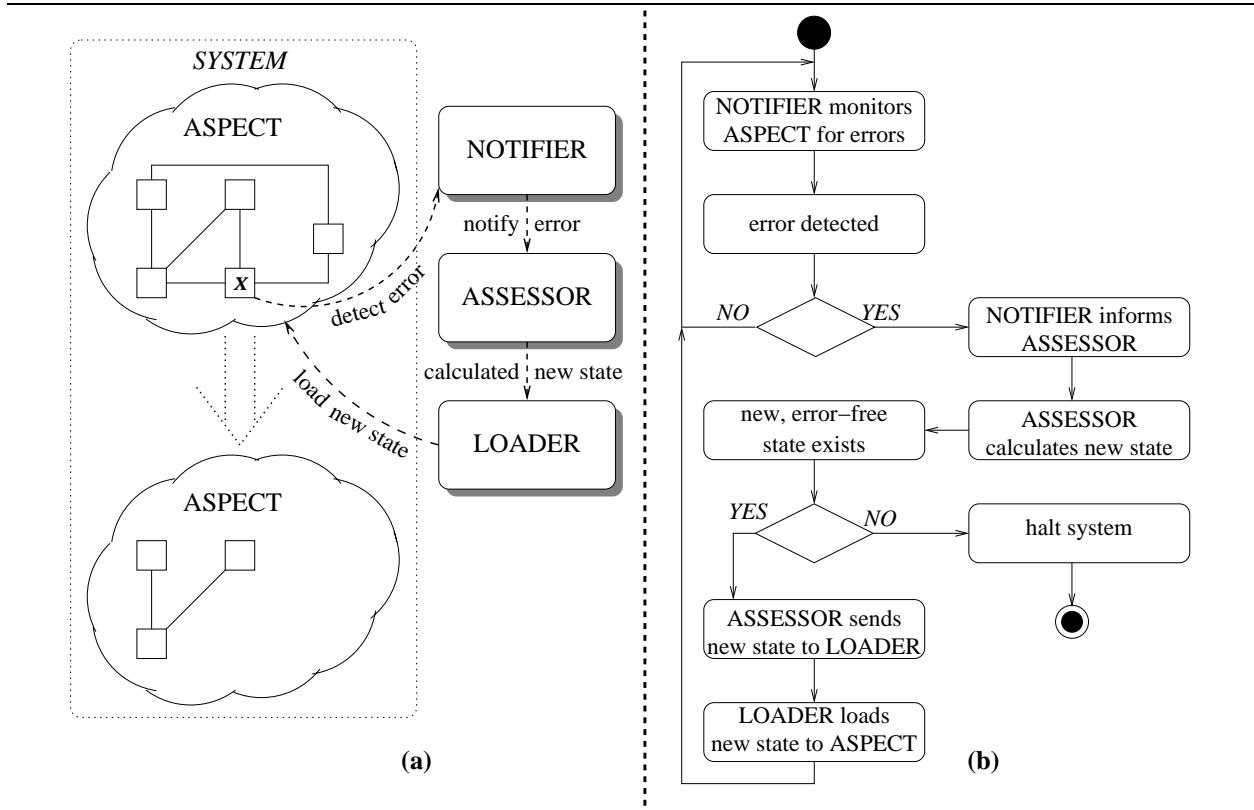


Figure 1: The structure (a) and the activity diagram (b) of the State Decrement pattern.

Figure 1a provides an intuitive illustration of the structure of the State Decrement pattern. The *notifier* entity monitors a given *aspect* of the system for errors and when such occur, it notifies the *assessor*. When the *assessor* is activated, it calculates a new, error-free state (if such exists) for the *aspect* and communicates it to the *loader*, which is responsible for loading the new state onto the *aspect*. Notice that, in the depicted example, the error (marked with *x*) on one module of the *aspect* led the *assessor* to calculate a new state of the aspect where not only the failed module is removed (the rightmost module of the *aspect* is also removed). The logic implemented by the *assessor* may require that more modules than the failed one have to be removed. This is further elaborated in the other three patterns presented in this paper. Figure 1b contains the activity diagram that describes the activities of the State Decrement pattern that include the notification of the occurrence of an error, the calculation of the new, error-free state and its installation in the system.

2.6 Example Resolved

The WM is designed as a component-based system, where each of the additional features (i.e. 3D shadows, cursor animation, window frames and menu-bars) is provided by a different component. An error detection mechanism plays the role of the *notifier* by monitoring these components for errors. When any of those components experience an error, the fact is communicated to a separate component that implements the *assessor* logic and which decides to remove from the system the components affected by the error. The *loader* role is played by the configuration subsystem of WM, which provides the capability of hot-plugging and unplugging components to the system. The *assessor* instructs the WM configuration to remove the affected components. For the sake of this example we consider that the only component affected by the error is the one that provides 3D shadows. After its removal, any window that needs to be redrawn on the screen (e.g. following a moving or resizing operation) does not have a 3D shadow. Despite this decrease in the functionality provided by WM, the system continues operating and provides all other functionalities (i.e. basic operations on windows plus cursor animation, window frames and menu-bars).

2.7 Implementation

The implementation of the **State Decrement** pattern consists roughly of the following steps:

- Specify the states of the *aspect* that are acceptable in an execution of the system. This step must be done in early design stages of the system since it may significantly affect the overall design.
- Ensure the existence of the *loader*, i.e. that the system has the capability of switching from one such state to the other, without halting its execution.
- Decide what mechanisms will be used for detecting and reporting errors, i.e. the ensemble of mechanisms that correspond to the *notifier*. This step determines which errors will trigger a graceful degradation of the *aspect*.
- Implement the *assessor* functionality, which decides what is the new, error-free state of the system after the occurrence of an error reported by the *notifier*.

The specification of the acceptable states that an *aspect* may take is specific to the application domain of the system and to the system aspect itself. Although a system aspect may have many possible states when not constrained by anything but the engineering properties of the system, a subset of these states may be unacceptable in a specific operational environment where the system executes. Even more, a subset of the states can be potentially acceptable in a given operational environment, but when specific conditions in the operational environment hold these states are not acceptable.

For example, in many environments where the window manager WM could be deployed (e.g. desktop and laptop computers), it would be able to survive an error that disables its capability to create new windows. However, when the WM is deployed on most of contemporary smartphones it is not able to survive the same error simply because the majority of user actions on an active window requires user confirmation, which is requested on a newly created pop-up window. Without the capability to create new windows, soon all the applications running in existing windows will crash because it will not be possible to create new

windows and request confirmation for the action that the user attempted on them. Those facts complicate the implementation of the *assessor*.

The implementation of the *notifier* depends on the events that it detects and reports. In practice, the *notifier* may not be a single entity but rather a set of entities which cooperate to provide error detection and notification, as well as reporting of other types of unsolicited events (e.g. power and temperature alarms).

The implementation of the *loader* usually requires (dynamic) reconfiguration capabilities. In certain cases, system design may integrate state switching as part of its “normal” functionality, which means that no reconfiguration mechanism is necessary but only employing this specific functionality is required for the state switching.

2.8 Consequences

The **State Decrement** pattern has the following benefits:

- + Compared to fault tolerance design patterns [16, 17, 19], the **State Decrement** pattern has lower costs in terms of system complexity and time and space overhead during system execution, while it ensures the continuation of a gracefully degraded system execution in the presence of errors. As such, it offers an appealing alternative to the designers of systems that cannot afford the costs of fault tolerance, yet they require to survive errors.
- + The use of the **State Decrement** pattern does not preclude the use of fault tolerance patterns [16, 17, 19] in the system design. In fact, the **State Decrement** pattern may share with fault tolerance patterns the error detection capabilities, and deal with those errors that fall beyond the scope of the application of the fault tolerance patterns. The errors that can put the system out of order can be intercepted and repaired by fault tolerance mechanisms, while those events that have less significant impact on the system execution can lead to the graceful degradation of some of its aspects without causing the system to crash. The resulting system would be able to tolerate a certain class of errors (those for which the fault tolerance patterns are used) and survive another class of errors (those for which the **State Decrement** pattern is used).
- + The **State Decrement** pattern does not introduce any time overhead to error-free system executions, unlike some fault tolerance patterns for checkpoint-based rollback recovery [17] and log-based rollback recovery [19].
- + Gracefully degrading to a lower, error-free state has the positive side-effect of stopping the error propagation.

The **State Decrement** pattern imposes also some liabilities:

- Applying the **State Decrement** pattern in the design of a system increases the complexity of that design. In the most favorable case, when the system has already error detection capabilities associated with some fault tolerance mechanism and dynamic reconfiguration capabilities too, the increase in the complexity of the system by **State Decrement** pattern is due to the introduction of the *assessor* entity. In the worst case, the **State Decrement** pattern adds to the design complexity of the system the cost of the *notifier* and *loader* entities, in addition to the cost of the *assessor*.

- To enable graceful degradation of a system aspect, that aspect must be modular. This implies that the aspect must be design in a way that it can have different states during the execution of the system. This is not feasible for all system aspects. Hence, graceful degradation has limited application.
- Often, bringing the system to a lower state implies dynamically reconfiguring the system during its execution. Dynamic reconfiguration mechanisms are shown to be costly to implement and they affect the have significant implications on the design of the system and its performance during the reconfiguration time (e.g. see [2]).
- A system cannot gracefully degrade beyond a certain number of successive occurrences of errors. In fact, most imlementation of various graceful degradation flavors allow a couple (or few of them) of successive degradation before the system crashes. That, along with the fact that a gracefully degraded system has lower qualities than its “normal” counterpart, implies that graceful degradation is a temporary solution to the problems caused by errors. To completely remove the effects of the error on the system, a restoration of the full-fledged state of the system must take place. Graceful degradation only allows the system to survive errors and possible wait for the opportune moment to engage the restoration process.

2.9 Related Patterns

The **State Decrement** pattern is realted to the Fault Containment patterns that have been presented in [18]. Also, it relates to the error detection patterns that have been presented in our previous work [16] and elsewhere [14]. Finally, the **State Decrement** pattern is tightly related to the next pattern presented in this paper, which elaborates on the way the *assessor* entity works.

3 Minimum Subtrahend

The **Minimum Subtrahend** pattern describes one possible design for the *assessor* entity of the **State Decrement** pattern presented in Section 2. In this pattern, the *assessor* calculates the new state of the system by simply removing the module of the state where the error was detected.

3.1 Example

A window manager WM is designed as a component-based system, which consists of one component that provides all the basic operations on windows (i.e. creating, moving, re-sizing, minimizing, restoring, maximizing, and destroying windows) each at a separate interface, plus a set of other components each of which provides one additional feature (e.g. 3D window shadows, window frames, cursor animation, menu-bars, etc) in a single interface per component. Each component executes in a separate thread inside the same process that hosts the entire WM system. The **State Decrement** pattern has been applied on WM to allow it to degrade gracefully on errors cause services to crash or compute wrong results. An error detection mechanism monitors each service invocation for errors and plays the role of the *notifier* entity. The dynamic reconfiguration capabilities of the system play the role of the *loader* entity. Only the *assessor* entity needs to be designed, and it required that when it calculates the new state it does not cause any unnecessary loss of functionality to WM.

3.2 Context

The context, in which the **Minimum Subtrahend** pattern can be applied to calculate the new, gracefully degraded state of a system after an error occurrence, is defined in terms of the following invariants:

- The **State Decrement** pattern has been applied to the system design. The system aspect that can be gracefully degraded has more than one state acceptable in the system execution. Also, there is a *notifier* that provides information about the occurrence of errors and a mechanism implementing the *loader* functionality as outlined in Section 2.5.
- The cost of engaging the loader mechanism is relatively small (e.g. it does not require the whole system to suspend its execution).
- The system cannot afford to suspend, or slow down, for long its execution while the new state is calculated and loaded onto the system.
- The *unit of failure* (i.e. the minimum entity of the system on which a failure can be observed) is well-defined.

The first context invariant indicates that the **Minimum Subtrahend** pattern can be applied in the context resulting by the **State Decrement** pattern. The next two invariants refine that context to represent the case where the loader mechanism is cheap to use and meets the need of the system to suffer minimum delays in its execution during reconfiguration events. Finally, the last invariant indicates a knowledge of the system entities that fail atomically, i.e. the minimum part of a system directly affected by an error. The definition of the unit

of failure is based on the structure of the system as well as on the types of errors that the system must survive by gracefully degrading its functionality.

For example, in a component-based system the candidate entities for the definition of the unit of failure are the hosts, the components, the interfaces, the operations inside an interface, the bindings, and the instances of a binding (i.e. the individual uses of an established binding). Transient network errors that may cause data to be dropped while on transit, affect only the instances of a binding; a failed service request due to such an error may succeed if re-tried over the same binding at another instance in time. Permanent network errors, e.g. caused by broken physical connections, affect only bindings; the same operation for which an error occurred over one binding, can be successfully invoked over another binding. Some software bugs, e.g. that may corrupt the static data of a function, affect only individual operations inside an interface; while one operation fails due to corrupted static data, other operations inside the same interface continue to function correctly.

Following the same reasoning, errors that cause the failure of socket ports affect only individual interfaces of some component; while none of the operations inside an interface that is mapped to the failed socket port will be accessible, operations in other interfaces mapped to different socket port continue to function correctly. Errors that cause entire process to fail affect only the components that execute within the failed process; while an entire component may fail due to an error that crashes the processes inside which it executes, other components running in different processes inside the same host continue operating correctly. Finally, the crash of an entire host would cause all the components deployed inside that host to fail, but it will not affect other components that are part of the system, which are deployed in other hosts.

N.B.: The statement that an error directly affects only certain system entities, expresses the fact that the immediate consequence of the error in question is the failure of the affected system entities, e.g. the crash of a host causes the failure of all components deployed in that host. However, the failure of those components causes the corresponding functionality to disappear from the system, which is a fault in the system composition. This fault may lead to other errors on those components in the system that try to invoke the functionality that has disappeared. Subsequently, those new errors may cause the invoking components to experience new failures. These new errors are different from the initial error, although they are causally related to it according to the ... \rightarrow *fault* \rightarrow *error* \rightarrow *failure* \rightarrow *fault* \rightarrow ... chain of error propagation [1, 10].

3.3 Problem

In the above context, a problem that arises is the following:

How to decide which will be the new state of a system after the occurrence of an error?

The **Minimum Subtrahend** pattern solves this problem by balancing the following forces:

- Removing from the system the entity that corresponds to the unit of failure on which an error is detected, results in a new state that does not contain the occurred error.
- Removing only the system entity that corresponds to the unit of failure on which an error is detected, may introduce a new fault in the system due to the absence of the functionality delivered by the removed entity.

- Calculating the exact impact of an error on a system state, i.e. identifying all the system entities that are directly or indirectly damaged by the failure caused from that error, can be very expensive and often impractical.
- After an error, calculating the new state by removing only the system entity that corresponds to the unit of failure on which the error is detected, is simple and fast.
- In many systems that process dynamically changing input from their environment (e.g. systems that interact with a human user), it is not possible to be certain that a system entity that depends on the functionality provided by another system entity will actually use this functionality during some system execution. Consider the WM example presented earlier: if the window creation functionality fails as soon as the system is started up, then it would not be possible to start any application since the attempt to create its graphical container would fail. However, if the window creation functionality fails at some point in the execution of the system after all applications that the user needs have been started, it is possible that the user uses the system without needing to invoke the window creation functionality.

3.4 Solution

After an error, the new state of the system is calculated by removing from the state in which the error was detected the system entity that corresponds to the unit of failure where the error was detected.

N.B.: The new state can be inconsistent, i.e. there may be system entities in the new state that rely on the functionality provided by the system entity that has been removed. This is acceptable in the suggested solution. The system entities that had invoked the removed functionality when the error occurred, and those entities that may attempt to invoke the removed functionality later in the same system execution will experience a new error, that of the absence of the functionality they have invoked or attempt to invoke. Some systems may possess a fault tolerance mechanism that deals with these errors, and other system may consist of entities robust enough to continue executing correctly despite these errors. For the rest of the systems, the new errors are reported to the *assessor*, which instructs the removal of the system entity where the error occurred, as described in this solution. After a number of such iterations, all the system entities – and only those entities – that have been damaged by the propagation of the initial error will be subtracted from the system.

3.5 Structure

The **Minimum Subtrahend** pattern does not introduce any new entities in the system where the **State Decrement** pattern has already been applied. It only elaborates on the logic based on which the *assessor* calculates the new, error-free state. We insist on the fact that the new state calculated by the *assessor* according to the **Minimum Subtrahend** pattern is “*error-free*”, since there are no errors reported on the units of failure that are part of that state. However, this does not mean that the new state is fault-free. In fact, it is likely that the removal of the functionality on which an error is detected cause a fault in the system composition, which will lead to an error when some entity in the new state of the system attempts to invoke the removed functionality. If the new error is reported by the *notifier* to the *assessor*, the graceful degradation mechanism will be engaged again, as discussed earlier.

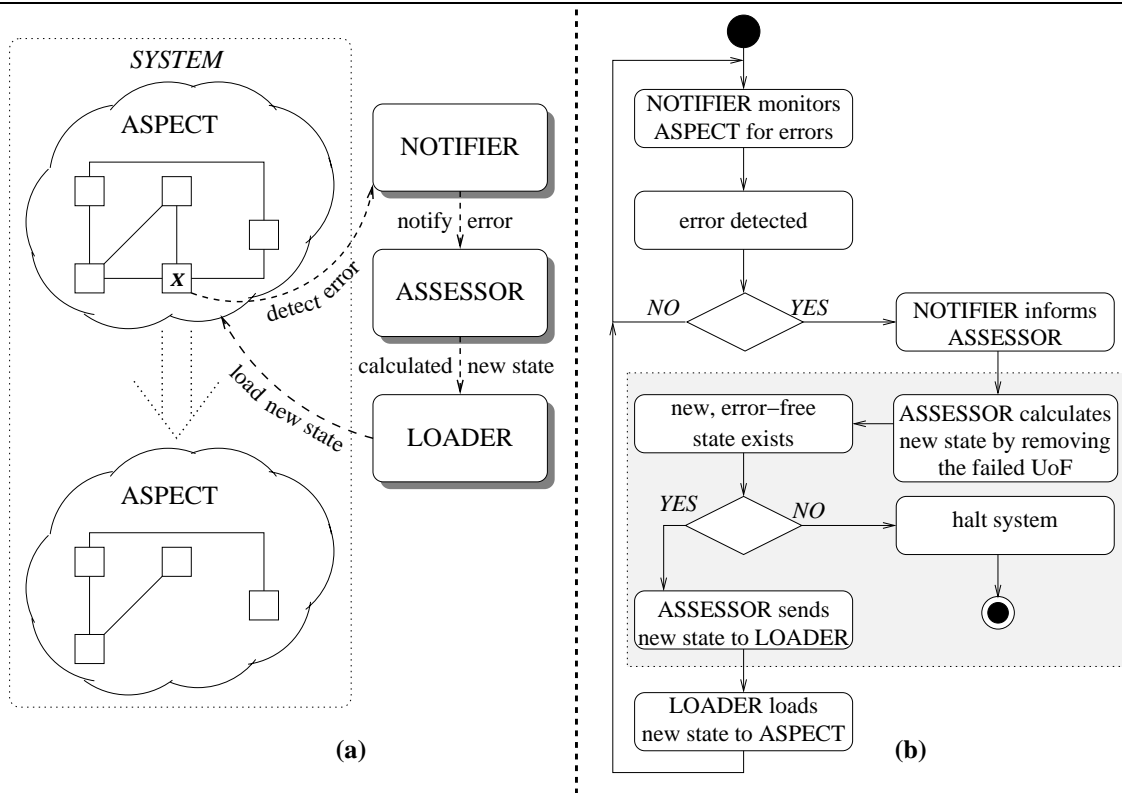


Figure 2: The structure (a) and the activity diagram (b) of the `Minimum Subtrahend` pattern.

Figure 2a updates Figure 1a to reflect the exact result of this logic on the calculation of the new state of the system after an error. In this figure, boxes depict units of failure and the error has occurred on the unit of failure marked with x . Similarly, Figure 2b updates the activity diagram of the `State Decrement` pattern from Figure 1b with a more precise description of the *assessor* activities as described by the `Minimum Subtrahend` pattern.

It is worth noticing that in practice, it is possible to simplify the shadowed part of the activity diagram in Figure 2b into a single activity, which is to instruct the *loader* to remove the system entity on which the error is detected. The resulting state is always error-free, as mentioned above, and the system will only halt if the removed entity is the last remaining entity in the system.

3.6 Example Resolved

The *assessor* entity, which is responsible to gracefully degrade the functionality of WM when errors occur, is designed according to the `Minimum Subtrahend` pattern. The unit of failure in the WM system is mapped to individual interfaces, and the error detection mechanism reports to the *assessor* an error and the identifier of the interface which provides access to the functionality where the error occurred. When the *assessor* is triggered by an error that occurred on interface X , it instructs the *loader* (i.e. the dynamic reconfiguration capabilities of WM) to replace the interface X by the NULL address in the system. As a result, subsequent attempts to invoke interface X result in an exception that indicates the lack of a binding.

When an error is reported on the only interface provided by the component that im-

plements the 3D shadows functionality, the *assessor* instructs the dynamic reconfiguration capabilities of WM to remove that interface from the system. Later, when some application attempts to create a new window, the window creation functionality attempts to invoke the 3D shadows functionality, but the invocation fails since the latter is not accessible any more. The window creation functionality is robust enough to continue operating despite the absence of the 3D shadows functionality (e.g. it can draw a window without any shadow) and the system continues operating correctly, though in a gracefully degraded state that does not offer any more 3D shadows.

A similar situation happens later, when the user of the system attempts to resize a window. The window resizing functionality attempts to invoke the 3D shadows functionality, but the invocation fails since the latter is not accessible any more. The window resizing functionality is not robust enough to continue operating without the 3D shadows functionality, and a new error occurs in the system. The *assessor* is informed about the occurrence of that error on the window resizing interface, and it instructs the dynamic reconfiguration capabilities of WM to remove that interface from the system. The system continues operating in a gracefully degraded mode, without any 3D shadows or the capability to resize existing or newly created windows.

3.7 Implementation

Provided that the functionality of the *notifier* and *loader* roles is already implemented in a system, the implementation of the *assessor* role as outlined by the **Minimum Subtrahend** pattern is a fairly straightforward process. First, the designer must identify the errors for which graceful degradation must be applied. Some error on critical system functionality may require the use of a fault tolerance mechanism that fixes the damages they caused and recovers the system in its full-fledged state. But errors on non-critical functionality are good candidates for being treated by a less expensive, graceful degradation mechanism.

The second step in the implementation of the **Minimum Subtrahend** pattern is the association of each of the errors identified above with their corresponding unit of failure. Different errors in a system may affect entities of different scale. For example, in a component-based system, an expired timeout on a given binding instance (i.e. individual invocation of an operation inside an interface) can be caused by an intermittent fault on that given binding instance and can be dealt with by retrying the same invocation over the same binding. Hence, it would be wiser to associate a single expired timeout to the individual binding instance where it occurred, and associate with the entire binding only the occurrence of, say, three successive timeouts on that same binding. It can be more appropriate to associate other types of errors, like exceptions reporting severe internal errors, to entities such as interfaces or components.

Once the errors, for which the graceful degradation mechanism will be engaged, are identified and associated to their appropriate unit of failure, the designer must specify how the new state calculated by the *assessor* will be communicated to the *loader*. When the error detection mechanism reports errors on the assumed unit of failure, this communication is straightforward: the new state is the state in which the error occurred minus the unit of failure on which the error is reported. In the WM example presented above, this situation appears when the unit of failure is a binding and errors are reported on bindings. However, in certain cases the error detection mechanism may report errors on smaller entities than what corresponds to a unit of failure. In the WM example, this situation appears when the unit of failure is an entire interface or component, but the error detection mechanism reports

errors on bindings. Such cases require that the *assessor* possesses the means that allow it to identify the unit of failure (i.e. the interface or the component in this example) based on the information communicated by the *notifier* (i.e. the binding on which the error is detected).

Another issue that requires attention when implementing the **Minimum Subtrahend** pattern is the identification and collection of “*garbage*”, i.e. system entities that consume resources without contributing to the correct execution of the system. Figure 3 provides the graphical illustration of an example where the new state calculated by the *assessor* may contain garbage. Assuming that the unit of failure is a component, depicted by a square in this illustration, and that the *notifier* reports errors on components, the error on the component marked with *x* on the left-hand side of Figure 3 causes the *assessor* to instruct the removal of that component. The resulting new state is depicted on the right-hand side of Figure 3, where component marked with *x* is no longer present, the bindings from components 2 and 4 to the failed component are broken, and component 3 is left disconnected from the rest of the system.

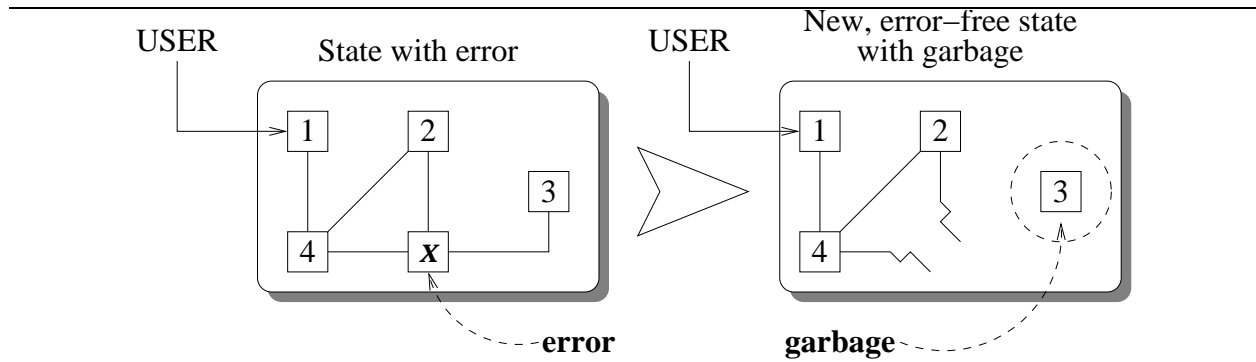


Figure 3: An example of the *assessor* calculating a new state that contains garbage.

When looking at the new state in Figure 3, it appears obvious that component 3 is garbage, since it is disconnected from the rest of the system and from the environment of the system. However, it is far from trivial to distinguish between the case that component 3 is garbage and the case where the last active client of component 3 has disconnected from it and component 3 waits for new client to bind to it. The **Minimum Subtrahend** pattern does not provide a solution for the issue of garbage collection, but the system designer must consider how to deal with it since it may cause the consumption of the system resources by garbage. This case could appear, for example, when the unit of failure is a binding, and bindings are removed from the system when they fail without any provision of removing components that are left unbound from the rest of the system and its execution environment.

3.8 Consequences

The **Minimum Subtrahend** pattern has the following benefits:

- + The design overhead is low, since the complexity of the *assessor* is low and regards the process of identifying the unit of failure where the error occurred and instructing its removal from the system. In case the error detection mechanism reports errors on system entities that correspond to units of failure, this complexity is the lowest that can be achieved by the different methods for gracefully degrading system functionality (e.g. see [20], [22], and [23]).

- + The decision on the new state of the system after an error takes little time, the time necessary to identify the system entity that corresponds to the unit of failure where the error occurred.
- + Once the iterative employment of the graceful degradation is completed and the system is in a consistent state, the part of the system state that has been removed is the minimum one that could be subtracted from the full-fledged system state and allow the system to survive the initial error.
- + The exact impact of different types of errors on the system state at various moments of the system execution does not have to be known. Rather, during its iterative employment the graceful degradation mechanism will gradual remove one after the other the system entities that get affected by the initial error.

The **Minimum Subtrahend** pattern imposes also some liabilities:

- The time necessary for the system to stop the error propagation related to the faults created by the removal of functionality by the graceful degradation mechanism, is not known in advance. As a consequence, after a severe error which the system cannot survive, it takes longer for the system to halt its execution than if it would halt immediately on error detection, and this additional time is spent in gradually removing one after the other the system entities as they fail. Such cases bear similar inconveniences to the system execution as does the “*domino effect*” in rollback recovery (e.g. see [13] and [17]).
- When the *assessor* is designed according to the **Minimum Subtrahend** pattern, it may leave garbage in the new state it calculates after an error, which consumes system resources without contributing to the correct execution of the system.
- The **Minimum Subtrahend** pattern does not apply to certain types of systems that do allow any period of inconsistent state during their execution.

3.9 Related Patterns

The **Minimum Subtrahend** pattern complements the **State Decrement** pattern presented in Section 2, for which it provides an elaborated solution to the way the *assessor* entity functions.

References

- [1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.
- [2] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 35–42, May 1998.
- [3] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

- [4] Y.-C. Chen, K. Sayood, and D.J. Nelson. A Robust Coding Scheme for Packet Video. *IEEE Transactions on Communications*, 40(9):1491–1501, September 1992.
- [5] V. Cherkassky and M. Malek. A Measure of Graceful Degradation in Parallel-Computer Systems. *IEEE Transactions on Reliability*, 38(1):76–81, April 1989.
- [6] K. Cheung, G. Sohi, K. Saluja, and D. Pradhan. Organization and Analysis of a Gracefully-Degrading Interleaved Memory System. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 224–231, June 1987.
- [7] P.P. Dang and P.M. Chau. Robust Image Transmission over CDMA Channels. *IEEE Transactions on Consumer Electronics*, 46(3):664–672, August 2000.
- [8] M.P. Herlihy and J.M. Wing. Specifying Graceful Degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.
- [9] G. Lafruit, L. Nachtergaele, K. Denolf, and J. Bormans. 3D Computational Graceful Degradation. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 3.547–3.550, May 2000.
- [10] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.
- [11] Y.-H. Lee and K.G. Shin. Optimal Reconfiguration Strategy for a Degradable Multi-module Computing System. *Journal of the ACM*, 34(2):326–348, April 1987.
- [12] S.R. Mahaney and F.B. Schneider. Inexact Agreement: Accuracy, Precision, and Graceful Degradation. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.
- [13] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.
- [14] K. Renzel. Error Detection. In *Proceedings of the 2nd European Conference on Pattern Languages of Programs (EuroPLoP)*, June 1997.
- [15] F. Saheban and A.D. Friedman. Diagnostic and Computational Reconfiguration in Multiprocessor Systems. In *Proceedings of the 1978 Annual Conference*, pages 68–78, December 1978.
- [16] T. Saridakis. A System of Patterns for Fault Tolerance. In *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 535–582, June 2002.
- [17] T. Saridakis. Design Patterns for Checkpoint-Based Rollback Recovery. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP)*, September 2003.
- [18] T. Saridakis. Design Patterns for Fault Containment. In *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 493–519, June 2003.
- [19] T. Saridakis. Design Patterns for Log-Based Rollback Recovery. In *Proceedings of the 2nd Nordic Conference on Pattern Languages of Programs (VikingPLoP)*, September 2003.

- [20] T. Saridakis. Graceful Degradation for Component-Based Embedded Software. In *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE)*, pages 175–182, July 2004.
- [21] T. Saridakis. Towards the Integration of Fault, Resource, and Power Management. In *Proceedings of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP-2004)*, pages 72–86, September 2004.
- [22] T. Saridakis. Surviving Errors in Component-Based Software. In *Proceedings of the 31st Euromicro Conference on Software Engineering and Advances Applications, Component-Based Software Engineering Track*, August-September 2005.
- [23] C.P. Shelton and P. Koopman. Improving System Dependability with Functional Alternatives. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, pages 295–304, June 2004.
- [24] A. Thomasian and A. Avizienis. A Design Study of a Shared Resource Computing System. In *Proceedings of the 3rd Annual Symposium on Computer Architecture*, pages 105–112, January 1976.
- [25] S. Yajnik and N.K. Jha. Graceful Degradation in Algorithm-Based Fault Tolerant Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):137–153, February 1997.
- [26] G.V. Zaruba, I. Chlamtac, and S.K. Das. A Prioritized Real-Time Wireless Call Degradation Framework for Optimal Call Mix Selection. *Mobile Networks and Applications*, 7(2):143–151, April 2002.