Technical Note PR-TN 2005/00451

Issued: 01/2006

Introduction to Software Fault Tolerance

Concepts and Design Patterns

R.T.C. Deckers; P.L. Janson; F.H.G. Ogg; P. J. L. J. van de Laar
Philips Research Eindhoven

Authors' address

R.T.C. Deckers	HTC31.2	robert.deckers@philips.com
P.L. Janson	HTC31.3	p.janson@philips.com
F.H.G. Ogg	HTC34.5	felix.ogg@philips.com
P. J. L. J. van de Laar	HTC31.2	pierre.van.de.laar@philips.com

Title:	Introduction to Software Fault Tolerance; Concepts and Design Patterns
Author(s):	R.T.C. Deckers; P.L. Janson; F.H.G. Ogg; P. J. L. J. van de Laar
Reviewer(s):	Reinder Haakma, Sjir van Loo, Rob van Ommering, Michel Meeuws, Chritiene Aarts, Gerard van Loon, Louis Stroucken, Ben Pronk
Technical Note:	PR-TN 2005/00451
Additional Numbers:	version 34
Subcategory:	-
Project:	REPA, TRADER
Customer:	Philips CE, Philips PS, Embedded Systems Institute

Keywords: Fault tolerance, error detection, software fault tolerance, error recovery, design patterns, error, error masking, dependability, failure, redundancy

Abstract: Software fault tolerance is a means to attain dependability. Dependability of a system is its conformance to user expected behavior. System failure, caused by faults in hardware or software, is avoided, recovered or hidden.

Both software fault tolerance and dependability are explained, as well as multiple views on the architectural consequences of their application in systems.

Furthermore, we present an inventory of software fault tolerance design patterns, as well as compositions thereof.

Conclusions: Software fault tolerance is derived from (older) hardware fault tolerance, but there is a prominent difference: traditional hardware solutions like spare parts cannot be applied straightforwardly.

We could not find specific HVE fault tolerance research. We see existing technology being applied to the crucial parts of HVE systems. When applied all-over the system, solutions may require resources beyond today's HVE practice. Moore's law is changing that balance.

New methodologies and rich Operating Systems, are currently increasing awareness of fault tolerance issues.

We expect off-the-shelve fault tolerance functionality to become available within Operating Systems, with Linux leading upfront.

Contents

Introduction

This document is an introduction to *Software Fault Tolerance*. We mean tolerance to software design faults and faults in the environment of the working software system. Next sections of this chapter explain the objective of this document, the intended audience, the problem we are trying to solve, the solution approach, and the background of the work. Chapter 2 is an introduction to the concepts of dependability. In Chapter 3 we focus on *Software Fault Tolerance*. Chapter 4 introduces our inventory of design patterns for fault tolerance, and the template used to describe them. Chapters 5, 6, 7, and 8 present the patterns in detail. Finally, in Chapter 9 we discuss our conclusion, open issues and future work.

Appendix A contains the result of a brainstorm session for new fault tolerance mechanisms. Appendix B presents a meta-model of the basic dependability concepts, following [Laprie].

1.1. Objective

After reading this document, one should understand the concepts and terminology of software fault tolerance. It should help in selecting (and creating) proper fault tolerance mechanisms for an architecture. The mechanisms are described in the form of patterns such that they can be used in a uniform way.

We do not claim to provide all required knowledge to realize a fault tolerant system. That is beyond the purpose of this document.

1.2. Intended audience

This document is intended for any software engineer looking to understand the principles of software fault tolerance. Architects can benefit from the inventory of fault tolerance mechanisms for their architectures. Researchers can use this document to identify gaps in the available technology landscape.

Since this document is non-classified it can be freely distributed to non-Philips readers. However, we are obliged to warn the reader twofold:

1. Pieces of text have been copied from other reports and books field without explicitly quoting the reference. Especially from [Laprie], [FTBOOK], [BOOK], [Lyu 96], [NASA], [Anderson81], and [Postma]
2. We sometimes refer to documents that are only accessible to Philips readers.

1.3. Problem description

Designing fault free products is practically impossible in many of today's system developments. Therefore, system architectures have to assure a certain level of fault tolerance against both design faults and faults in the system environment. Although much literature on fault tolerance exists, we did not find a short introduction that covers the main concepts, nor an inventory of Fault Tolerance mechanisms. Furthermore, it is not

clear how much research is needed and in what way Philips products can become fault tolerant.

Most ideas and concepts in the existing literature are not directly applicable to the domain of high volume electronics (HVE). They are mostly suited for safety and mission critical systems for NASA and e-commerce applications. The constraints typical for HVE products– such as bill of material (BOM) and development costs - are ignored in existing literature on fault tolerance.

The latest trend in software development for HVE is to buy 3rd party software (e.g. Commercially Of-The-Shelve software, COTS) to realize non-differentiating[1] functionality. The fault (in-)tolerance of a COTS sub-system in a product may influence the fault tolerance of the whole product. Hence, fault tolerance mechanisms to isolate and contain faults of COTS[2] are very welcome.

1.4. Solution approach

This report summarizes articles and books about fault tolerance, and our interviews with people that are active in fault tolerance. Parts of this document (including some graphics) were copied from these sources.

We brought together the mechanisms from these sources in a consistent inventory, made them comparable, and underlined any consequences of their usage in HVE.

1.5. Work background

This document is the result of a co-operation between two research projects:

> the REPA project at Philips Research, *Recovery Enabled Product Architectures,*
> *&*
> the TRADER project at the Embedded Systems Institute (ESI) *Television Related Architecture & Design to Enhance Reliability*.

These projects have joined forces in exploring the field of fault tolerance. In the future, we expect more architects to go through the same phase. This document should help those architects, providing them with an explanation of basic concepts and an inventory of fault tolerance design patterns. Some design patterns are based on implementations of the SPACE4U research project for ITEA.

1.6. Acknowledgements

Special thanks to:

- Rico Kind for providing SPACE4U documentation and for numerous discussions about fault tolerance, while he reviewed our design pattern descriptions.
- Merijn de Jonge, for providing REPA design patterns and for his feedback.

[1] "Non-differentiating" here means, functionality that is available in Philips' competitor's devices too.

[2] Research on 3rd party software, and the integration thereof, resulted in the TiPSI checklist. Based on real experiences of Philips development teams, it aims to minimize project risks of working with COTS. [TIPSI]

- Ben Pronk, Jozef Hooman, Louis Stroucken, Rob Golsteijn & Iulian Nitescu for their feedback.

1.7. Abbreviations

ESI	Embedded Systems Institute
FT	fault tolerance
REPA	“Recovery Enabled Product Architectures”, project at Philips Research
TRADER	“Television Related Architecture & Design to Enhance Reliability”, project at Embedded Systems Institute.
BOM	Bill of Materials; common term to denote the joined material costs of production of a consumer electronics device.
HVE	High Volume Electronics; the market segment Philips Consumer Electronics and Philips Semiconductors are in.
SPACE4U	ITEA project “Software platform and component environment 4 you”. http://www.hitech-projects.com/euprojects/space4u/

2. Introduction to dependability

This chapter gives an introduction to *dependability* and related concepts. In the context of system development, we consider *fault tolerance* as a means toward increasing the *dependability*. Therefore, we start by explaining some prime concepts such as Fault, Error, Failure, and of course dependability itself.

Dependability is commonly divided into sub-qualities that we present in Section 2.2. Section 2.3 presents the means to attain dependability. We will end this chapter with a summary (Section 2.4).

2.1. Dependability concepts

We consider the leading paper in dependability research to be [Laprie]. It provides a definition of dependability concepts, aspects and means. We follow its terminology, because it suffices for the purpose of this document. We have only described the *main* concepts with some of our own comments, in this chapter.

For background information on concepts employed in the field of Dependability we refer to [Laprie]. In addition, we present models that relate these concepts to each other visually, in Appendix B. This provides some support to the reading of [Laprie].

Figure 1 shows the relationships between the three prominent concepts of dependability: Fault, Error and Failure. The definition of dependability uses these. We define them in the following sections.

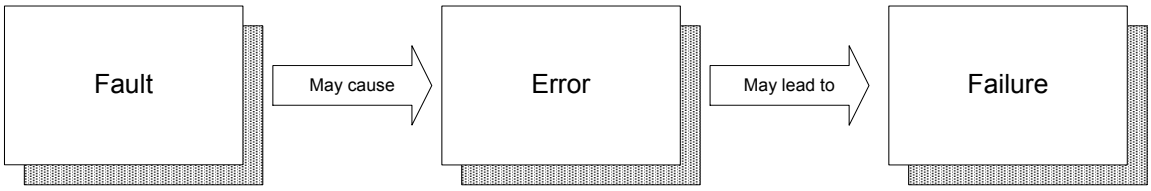


Figure 1: Relationship between Fault, Error and Failure.

2.1.1. definition: Fault

The adjudged or hypothesized cause of an error.

A fault is the result of a mistake made in the development of the system, or in the environment of the system in operation. Such a mistake is the potential cause of an error. Programming faults can be introduced during development, ranging from wrong or forgotten requirements to typos in the source code. Basically, a fault means that someone[3] (apparently) made a mistake, which could also be an unforeseen interaction or a defective part (hardware or software).

Faults are *dormant* and can become *active*. When a fault becomes *active*, it causes an *error*, which may influence the system's dependability.

[3] Can also be a group of people or an external system that is made by people.

2.1.2. definition: Error

The part of the total system state that may lead to its subsequent service failure.

An error is the manifestation of something actively going wrong in the running system. Often errors lead to new errors (*propagation*), which eventually may lead to system *failure*. It is often very difficult to detect the first error in a chain of errors. For example: a programming fault may lead to a memory leak, which may lead to a memory shortage, which may lead to an observable application crash. The original memory leak may be difficult to detect.

2.1.3. Definition: Failure

An event that occurs when the delivered service deviates from correct service.

A service fails either because it does not comply with the functional specification (including timeliness) or because this specification did not adequately describe the system's function.

A failure is simply an error that is observed from the system environment, through any of the system's interfaces. This does not mean that all failures are observed, because the failure is often acceptable (e.g. a small unnoticed artifact in a TV's image). What the exact correct service is, is not always clear. The correct service may not be specified (or specifiable) in practice.

The *severity* of a failure is proportionate to the impact on its environment.

2.1.4. Definition: Dependability

The ability to avoid service failures that are more frequent and more severe than is acceptable.

Dependability is how well you can count on the system. This user perspective on dependability means that it is a subjective property. What one person considers dependable may not be dependable to another person. This implies that dependability requirements cannot be quantified without agreement on the user perspective. Because the definition of dependability refers to the definition of failure, dependability is not determined by the functional specification of the system, nor by some official document, since that specification may be incomplete or incorrect with respect to the user's expectations.

There exist two main views upon dependability; the quantitative view and the qualitative view. In the quantitative view, the aim is make dependability measurable, using expressions like *Mean Time Between Failures* (MTBF), *Mean Time To Failure* (MTTF) and *Mean Time to Repair* (MTTR), but also maturity quantifications such as the number of reported faults in the software, or the percentile of passed tests.

In the qualitative view, the aim is to describe – rather than measure - aspects of systems. In this field we encounter terminology like *Fail Safe* and *single-point-of-failure*.

A *Fail Safe* system only has failures that have an acceptable safety impact, i.e. it will never pose a hazard to the user or the environment.

A *single-point-of-failure* is a fault that causes total system failure when it is activated. Often mission critical systems compensate for such faults, by incorporation of back-up systems and distribution of control.

2.2. Dependability attributes

Qualitative dependability is often detailed into sub-qualities called dependability attributes. These attributes are used to specify dependability requirements:

- **Availability**: readiness for correct service. (E.g. when a washing machine is in repair it is unavailable)
- **Reliability**: continuity of correct service. (E.g. a washing machine that may stop in the middle of a washing program is unreliable.)
- **Safety**: absence of catastrophic consequences on the user and the environment. (E.g. a washing machine is unsafe if one is electrocuted while opening the door.)
- **Integrity**: absence of improper system alterations. (E.g. the water in the washing machine may never boil.)
- **Maintainability**: the ability to undergo modifications and repairs. (E.g. a washing machine that cannot be dismantled to replace inner parts is not maintainable.)

In the context of security an additional attribute is:

- **Confidentiality**: absence of unauthorized disclosure of information (E.g. Usage pattern files of the washing machine of public figures should not be disclosed to tabloids.)

Architects need to design dependable products, balancing the above mentioned dependability attributes against their respective costs. Therefore, in the following section we will look at how to realize such dependability requirements.

2.3. Means to attain dependability

The means to realize the proper dependability of a system are categorized in:

- **Fault prevention:** preventing the occurrence or introduction of faults. This is any measure taken to prevent faults from being present in the system in the first place. Examples are reviewing, training of developers, applying design patterns. Other examples are methodologies such as *Defensive Programming* [McConn04] and *Test Driven Development* [BECK]. Fault prevention applies to development time.
- **Fault tolerance:** avoiding service failures in the presence of faults. These are measures to make the system robust against unspecified behavior of its environment (e.g. users or other systems), as well as measures to protect the system against errors caused by faults in its parts, both hardware and software. Fault tolerance applies to systems at run time.
- **Fault removal:** reducing the number and severity of faults. These are measures to repair a system or to prevent dormant faults from being activated (again). For example: software upgrading after sales. Fault removal applies to both development time and run time.

- **Fault forecasting[4]:** estimating the present number, the future incidence, and the likely consequence of faults. Typically development tools to analyze potential, injected faults and tools to localize faults can be regarded as fault forecasting means.

2.4. Summary

The field of dependability concerns *service failures*, which are caused by (possibly propagated) *errors* due to one or multiple *faults*, present in the system. Dependability can be measured *quantitatively* and expressed *qualitatively*. In the latter case by using the *dependability attributes*. To attain dependability we must handle problems at their root, the system faults. Faults can be *prevented* and *forecasted*, but they can also be *tolerated* (handled) and *removed*.

We consider the *tolerance* and (run-time) *removal* of system faults both part of fault tolerance, which we zoom into in the next chapter.

[4] Fault forecasting is sometimes referred to as fault prediction

3. Fault tolerance

Fault tolerance is the tolerance for system faults, and can include the runtime removal of them, with the purpose to improve dependability. In this chapter, we provide several views on fault tolerance. We have chosen these views to help the reader to improve his insight into the needs for and means of fault tolerance. After reading this chapter, the reader should be able to identify the relevant attributes of his own system.

In section 3.1 we look at fault tolerance from the perspective of the system context by distinguishing application domains. Section 3.2 introduces the notion of fault tolerance levels. Levels help in choosing the right degree of fault tolerance for a specific application. Furthermore, they allow the re-use of solutions developed for each level. Section 3.3 lists the approaches we found in literature to realize fault tolerance. Section 3.4 discusses the design aspects of fault tolerance.

Again, we will end this chapter with a summary (Section 3.5).

3.1. Goals depend on the application domain

The *application domain* of a system determines what the important dependability goals are. These goals help to design fitting fault tolerance and can set a reference for all applications in the domain. Some examples of application domains[5] with their goals (*in italics*) are:

- **Critical applications** require a high degree of confidence on the *correct* and *safe operation* of the computer system in order to prevent loss of life or damage to expensive machinery.
- **Long-life applications** require that computer systems operate as intended with a high probability when the *time between scheduled maintenance* is extremely long (e.g. on the order of years or tens of years).
- **Delayed-maintenance applications** involve situations where maintenance actions are extremely costly, inconvenient, or difficult to perform. For this reason the system must be designed to have a high probability of being able to continue operating *without requiring unscheduled maintenance* actions.
- **High-availability applications** require a very high probability that the system will be *ready* to provide the intended service when required. This type of systems allows frequent service interruptions of short duration.
- **Commercial applications** are typically less demanding than the abovementioned applications. The main use of fault tolerance in these systems is to *prevent nuisance faults* from affecting the *user's perception* of dependability. Also, more dependable products may in some markets have a higher sales price.

To test whether the dependability goals are achieved, one can express them quantitatively, such as defining a target mean-time-between-failure (MTBF) and target mean-time-to-repair (MTTR).

[5] based on [NASA]; Chapter 5.1

One system may contain multiple applications, each with different dependability goals. The designer could choose a different level of fault tolerance for each of the applications, as we will discuss in the next section.

3.2. Levels of fault tolerance may differ

We can discern different levels of fault tolerance. A designer must decide which level of fault tolerance is needed for his/her envisioned system. Different parts of the system may require different levels of fault tolerance.

It is not a straightforward task to discern fault tolerance levels within the scope of a design. Neither, can we provide a single solution. We will however, present an example based on [FTBOOK] 10.3.2, where fault tolerance for a distributed system is described.

Level 0: No tolerance to faults.

Level 1: Automatic error detection and system restart.

Level 2: Level 1 plus periodic check-pointing, logging and recovery of internal state.

Level 3: Level 2 plus persistent data recovery.

Level 4: Continuous operation without any interruption.

It is left to the architect to define the levels of fault tolerance applicable to his application domain.

3.3. Approaches to realize fault tolerance

Different approaches exist to realize software fault tolerance. A total approach might be a combination of these. The approaches we found are:

- **Application specific approach**: error detection and recovery is specifically designed for one application. It allows for the most accurate fault tolerance and allows to optimize for non-functional properties, like CPU and memory usage.
- **System level approach**: fault tolerance is implemented in system level components like the operating system. This approach may not be able to handle all errors.
- **Language approach**: Extending the design / programming language / environment with fault tolerance concepts, e.g., exceptions in C++ or a library of fault tolerance functions.
- **Meta-call approach**: fault tolerance is implemented by adding extra functionality to the implementation of function calls.

3.4. Design aspects of fault tolerance

Every fault tolerant design covers one or more of the below listed design aspects ([Lapri], [NASA], [Nelson 90], [Anderson81]).

3.4.1. Detection

Error detection is a prerequisite for other fault tolerance mechanisms. Detection is based on expectations and observability.

Detection can be *concurrent* or *pre-emptive*. Concurrent detection is done during normal service delivery. Pre-emptive detection suspends normal service to check the system for latent[6] errors and dormant[7] faults.

All detection is based on comparison. The expectations are compared with the observations. The expected information can be based on specifications, models, the equality of multiple kept versions etc. To give a few examples: comparing a value against its allowed range; comparison of the outputs of multiple implementations of the same functionality; comparison of the elapsed time of a computation with the specified worst case value.

The specific information to compare can differ for each error type. For example, timing errors are detected based on the expected duration of the actions and state-behavior errors are detected based on the design (state machine) of the software.

3.4.2. Diagnosis

Diagnosis identifies the cause of errors in terms of relevant parameters such as location and error type. In addition, the system can diagnose its own health, as additional input for recovery.

3.4.3. Containment

Complex systems generally contain boundaries to prevent errors in one part to propagate to other parts. Such isolated parts, *containment regions*, are usually arranged hierarchically throughout the structure of the system. Communication between these regions is controlled to reduce the risk of propagating errors via communicated data. Proper containment enables the designer to depend on a number of correctly operating components, which allow the system to continue its correct operation. Containment techniques also help to isolate a faulty component after detection of an error, to make the fault causing the error dormant (again). Traditional hardware containment regions are e.g. isolated memories, CPUs, clocks and communication links. Software containment is for example isolation of process spaces.

Recovery of a single region is only effective if the root-cause is removed and the error did not propagate to another region.

3.4.4. Recovery and Continued Service

Recovery restores a system to proper service by bringing it into an error-free state. For example this may be by restoring a previous state (roll-back).

In the presence of explicit containment regions (in 3.4.3. Containment), the recovery of a region must bring it back in a safe state and re-establish the relationship with its

[6] An error is *latent* when the detection algorithm has not detected it.

[7] *Dormant* faults are those faults that have not led to an error (yet).

environment. Recovery might result in side effects because the services provided by the region under recovery are temporarily unavailable.

3.4.5. Masking

Masking is the dynamic compensation for errors in such a way the system environment does not notice them. The system should pass only correct values to its environment, despite failure of its internal parts. For example, real-time systems often use masking, to hide errors in their output stream by degrading output quality[8]. Recovering from the error would otherwise take too long.

3.4.6. Repair or re-configuration

To prevent repeated activation of faults, some faults can be repaired. Repair can be replacement of affected components or rearrangement of the system's activities to prevent fault re-activation. The system may require to be re-initialized after a reconfiguration, which we consider an action of recovery.

3.4.7. Coverage

The effectiveness of any fault tolerance technique is measured in its *coverage*, which is essentially the probability of a system failure, given that an error occurs. Hence, fault tolerance coverage determines the dependability that can be achieved. A mechanism's fault tolerance coverage applies to its:

- Error handling: Which errors are handled?
- Fault handling: Which faults are handled, such that they will not be re-activated?
- Fault assumptions: Which faults are assumed to occur and which ones really occur?

In the context of redundancy, one must be aware of the *common mode failure:* A failure due to a single, specific fault causing all systems to fail. Multi Version Software is an approach that addresses this problem in software.

3.4.8. Redundancy

Definition:	**Redundancy in computer systems is the use of resources beyond the minimum needed to deliver the specified services.** [NASA p.31]

In computer systems, redundancy may manifest itself in both *space* and *time*. Redundancy in space can manifest itself in additional stored information, additional hardware, spare parts, etc. Redundancy in time can manifest itself in multiple executions of the same algorithms or additional algorithms. Redundancy can also manifest itself in explicit *design diversity*, i.e. the use of independent people, systems, environments, hardware, programming languages, or compilers (etc.) to provide redundant functionality.

[8] as in *Quality of Service*

Redundancy may orthogonally be present in system's hardware, software and data. In software fault tolerance, the above-mentioned *resources* are entities such as processing time, memory space, or bandwidth, to name a few. In this document we focus on such "soft" (non-hardware) resources.

Examples of redundant data in storage or communication:

- Duplicate, independent storage (backups)
- Encoding, like cyclic redundancy checks (CRC)
- Timing and acknowledgement in communication protocols

Examples of redundancy through the use of specification information:

- Verifying pre-conditions of calls to system functions
- Verifying the state and state transitions

Examples of redundancy through extra functionality:

- Independent systems providing the same functionality, at the same time
- Backup systems that become active when the systems fails
- Duplicate execution at another time or another space

3.5. Summary

The *goals* of fault tolerance depend on its application domain, ranging from concerns of loss of human life to maintenance, availability and commercial/competitive value.

Having chosen one or more goals, one can specify *levels* of fault tolerance for different parts of the system. Next, one can deliver that level of fault tolerance by adding fault tolerance to the application, or to the programming language, or to delegate it either to the platform (e.g. the Operating System) or to the language compiler (meta-call functionality). Surely, there exist even more options in this respect.

One must design how errors are *detected*, *masked* and *contained*. To *recover* from them one needs to *diagnose* which fault caused them, *repair* or *re-configure* the system and let it *continue* to deliver correct service. When designing fault tolerance, one must decide which faults and errors will be covered. Fault tolerance always requires redundancy, which comes at a cost.

4. An inventory of design patterns

In order to provide the reader with concrete examples of fault tolerance, the remainder of this document is dedicated to an inventory of fault tolerance design patterns. These patterns are categorized into chapters with

5. Atomic *detection* patterns
6. Atomic *recovery* patterns,
7. *Composite* patterns, composed from multiple other patterns, and
8. *Supporting* patterns.

In each of the chapters, we allocate a section to each pattern, described in a template that we introduce in section 4.1. Some overall disclaimers and attention points are mentioned in section 4.2.

In section 3.4 we showed that fault tolerance touches upon Error Detection, Diagnosis, Error Containment, Error Recovery, Masking, Repair, Coverage, and Redundancy. Of these aspects fault diagnosis, assessing coverage of fault tolerance technology and design redundancy are designer's (human) tasks. Fault tolerance *software* can detect errors (error detection), contain them (error containment), mask them (masking), and possibly recover the system from the error (error recovery). Therefore our inventory uses the concept of a *mechanism type* (error detection, error containment, masking, or error recovery)[9]. The aforementioned template (Section 4.1) contains this mechanism type.

[9] The inventory contains no repairing software patterns, hence no repair mechanism type is introduced.

4.1. Template for design patterns

To describe each design pattern we use the following template of aspects:

Name	(Semantic) identification of the pattern
Aliases	Any names under which we encountered a similar pattern in literature, interviews or common practice. Terms in this section are never a definition, but rather a means to let the reader find what he/she is looking for.
Mechanism type	One or more from *Error detection, Error recovery, Error masking, Error containment, Supporting.*
Description	One-liner describing what it is.
Mechanics	How the mechanism works and can be implemented. We provide the basic elements, their roles and their collaboration
Error type	The faults/errors that this mechanisms is meant for.
Benefits	Gain from embedding the mechanism
Required system context	The resources claimed by this mechanism and any other requirements posed upon its system context. This may for instance include availability of another mechanism.
Consequences	Relevant side effects, drawbacks or risks.
Side effects	Impact on the system (including its architecture) or the system development. Described in terms of *space*, *time* and *architecture*. It may thus cover timing aspects, development effort, maintenance, memory usage (RAM and ROM), and CPU cycles.
Drawbacks	Negative points
Risks	Crucial information to be attentive of when embedding the mechanism
Variations	Architectural[10] variations we suggest.
Alternatives	Substituent mechanism(s) with comparable purpose.
References	• Article, website, person or a project • Mention of an existing system that embeds the mechanism

[10] non-trivial variations. E.g. a trivial variation point of a timer is that its waiting period is configurable. A timer has an architectural (non-trivial) variation point in the sense that it can be embedded both in hardware and in software.

Wherever possible, patterns are illustrated with a UML diagram or a free-form figure.

4.2. Disclaimers and attention points

Before reading and using the design patterns, we wish to point out some issues.

In pattern descriptions we omitted the risk that by only inspecting the (regular) in-/output streams of a system one can never detect errors that do not affect a system's in-/output, e.g. memory leaks and resource hogging.

All design patterns require implementation effort and consume CPU cycles and memory when run. In particular cases, where resource consumption can be predicted more accurately, we mention such resource requirements in the *consequences*. In other cases, it is left to the reader to estimate the resource consumption applicable to his/her envisaged implementation.

In many cases, an aspect of a pattern can be considered a mere side-effect, a drawback or a risk. Depending on the particular context the reader has in mind for the pattern, he or she may disagree with our judgment.

Many mechanisms found in the literature may originate from application domains where resource constraints are less stringent than today's HVE domain constraints. However, the HVE domain evolves, hence some patterns may come within reach over time.

Readers should be aware that some patterns could be implemented inside a single, isolated system component, whereas other patterns require system-wide adaptations, often affecting many components.

We have developed the template ourselves. Therefore, if we omit any information that the reader considers fit, we may have ignored it deliberately, but it may also have escaped our attention. In the latter case we welcome your comments and suggestions.

5. Design Patterns for Error Detection

5.1. Time-out

Aliases: Deadline, alarm (clock), timer

Mechanism type: Error detection

Description: A time-out is the notification that an acitivity has missed its deadline.

Mechanics:
When an activity has a deadline, an observer sets a timer to signal an alarm at this deadline (see Figure 2). Either the activity finishes before the timer expires, or the activity finishes after the timer expires. If the activity finishes before its deadline, the observer cancels the timer and no alarm is sent. If the timer expires, the observer deduces that the activity has failed to meet its deadline and signals an error.

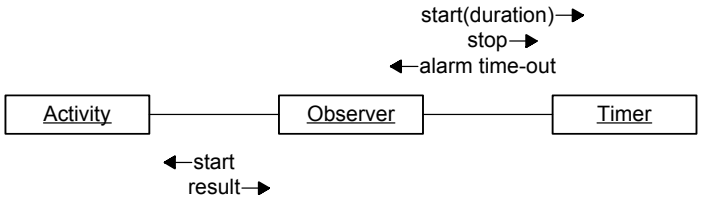


Figure 2: Time-out, UML Collaboration diagram

Examples:

1. A time-out is used to build a system that sends a message to its communication partner and must wait, say, 10 seconds for an answer. If the answering message is not received within 10 seconds, the system's observer must be alarmed to take appropriate action, such as re-sending the message.
2. A time-out is used to build a system that limits the duration of a complex computation. For example, a time-out can be set to ensure the system's responsiveness. (In this case the time-out is not the detection of an error.)

Error type: Timing errors; untimely response, no response

Benefits: A deadline can be enforced upon an activity.

Required system context:
A timer needs a (sufficiently accurate) clock, and must be independent of the activity. For example, the timer needs a high priority thread or interrupt to execute independently from delays and lock-ups caused by the activity it is associated with.

Consequences:

Side effect: The development process must accommodate for additional analyses of the requirements, to first find, and then set deadlines. This requires specialist domain knowledge.

Risk: The deadline might be hard to determine due to worst-case execution times, blocking, and interruptions. When the time-out duration is chosen too short, a time-out may occur erroneously. When the time-out time is chosen too large, unnecessary time is lost during errors.

Risk: Race conditions may occur timers that were cancelled send notifications.

Risk: Stale time-out notifications/flags must be ignored.

Variations:

- Time-out can be implemented without cancelling the timer: When the activity finishes, a flag is set. When the timer expires, the flag is checked. If the flag is not set, a time-out (error) is signalled. Note that in this variant, an expiring timer does not always signal an error.
- A timer can be provided in hardware and in software:
    - Hardware timers are generally more fine-grained (milliseconds) than software timers (from 0.1 sec up to minutes, hours, years) because the latter are implemented by regularly polling the system clock or follow a regular interrupt.
    - In every system the number of hardware timers is limited to a few, currently about 3. In modern operating systems however, the number of software timers is very large (100+).

Alternatives:

-

References:

- "Software Fault Tolerance: A Tutorial", Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616, Chapter 4.1.2 "Timing checks".

5.2. Heartbeat

Aliases: Watchdog, Alive signal

Mechanism type: Error detection

Description: A heartbeat notifies that a system is operational.

Mechanics:
A heartbeat is a regular (e.g. periodic) signal notifying that an observed system is fully or partially operational (alive). An observer monitors the heartbeat, and an error is detected when the observer no longer observes the expected heartbeat signal. A heartbeat is independent of the functionality of the observed system.

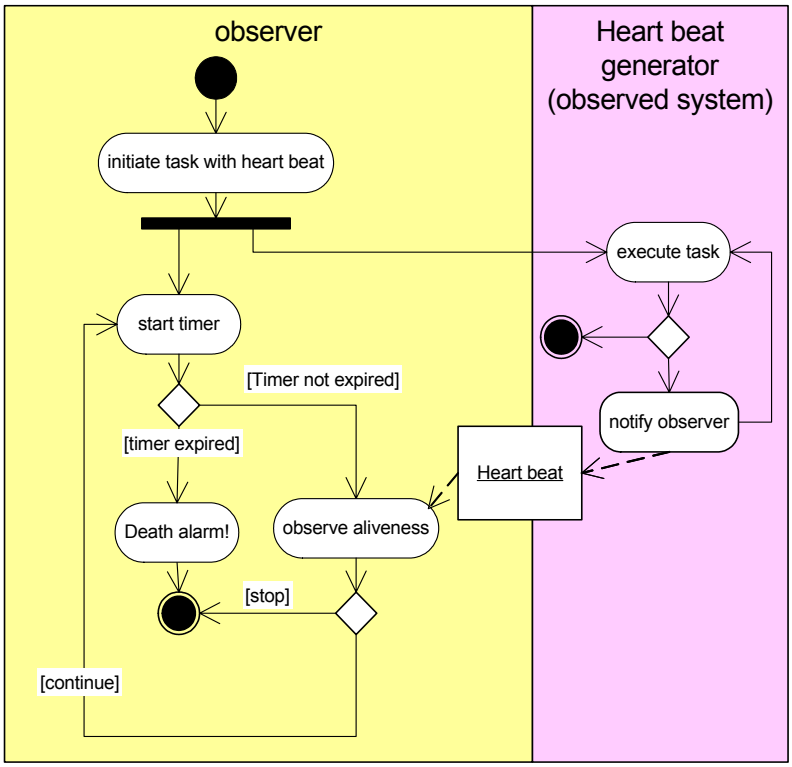


Figure 3: Heart beat, UML activity diagram

Error type: Deadlock, lock-up, no power

Benefits: Enables an observer to monitor that a system is operational.

Required system context: -

Consequences:

Risk: When the period (the maximum allowed time lapse between consecutive heart beats) – in timed heart beat systems - is set too short, the heartbeat mechanism erroneously signals to miss a beat (false positive). When the period is chosen too large, the missing of a beat is detected much later than it could have been detected, and valuable time is lost. The period of a heartbeat might be hard to determine due to worst-case execution times, blocking, and interruptions.

Risk: A situation can occur wherein the only function that the observed system is performing is its heartbeat, while the rest of the observed system is dead. Note that this demonstrates the independence of the heartbeat to the rest of the observed system.

Variations:

- A heartbeat can be based on either a push or a pull mechanism: The (sub)system can have its own heartbeat, or the observer can regularly request a notification from the (sub)system.
- A heartbeat can be implemented both in hardware and in software.
- An error can be reported after a number of heartbeats are not observed in time. In this case, the (sub)system can skip a beat.
- The observer can request only a notification from the (sub)system whenever “normal” communication, that proves the (sub)system is operational, is absent. This approach reduces the communication overhead.
- The most straightforward implementation uses a timer to detect the absence of a heartbeat. Yet, timers are not required. For example, in a system that contains two subsystems, which have the same heartbeat rhythm under normal conditions, we introduce a heartbeat counter for each subsystem. Initially, both counters have the same value. Whenever a heartbeat is observed, the corresponding counter is incremented and it is checked that the two counters do not differ more than a single beat. If the difference is larger than a single beat, either subsystem is not progressing at the right pace.

Alternatives:

Progress monitoring based on functionality and the passing of milestones. This approach requires modification of the (sub)system and domain knowledge.

References:

- *"Classes of Byzantine fault-tolerant algorithms for dependable distributed systems"*, Andre Postma, Proefschrift / Thesis, 1998, ISBN 90-365-1081-3, Chapter 1.2.5.
- *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 Chapter 4.1.2 "Timing checks".
- A television has multiple heartbeats.

- Article: ”*Watchdog Timers”*, Niall Murphy, http://www.embedded.com/2000/0011/0011feat4.htm
- On the Internet, load balanced HTTP servers are often equipped with a heartbeat wherein each server periodically sends a *ping* to its neighbour. If the *ping* is not answered, the sender can alarm an administrator.

5.3. Error Detecting Code

Aliases: Error Correcting Code, Checksum, Parity, CRC, Forward Error Correcting Code (FEC), Reed Solomon

Mechanism type: Error detection, Error recovery[11]

Description: Enriching data with encoded extra information to promote data integrity

Mechanics:
Encoding enriches a data item with a generated redundancy value, following a strict mathematical recipe (see

Figure *4*). The augmented data item is used by the reader to validate the received information. Decoding of a validated augmented data item consists of applying the reverse mapping of the recipe, to decode the original data item. If validated, the read data-item is trusted, but if the reverse mapping cannot be applied (i.e. if invalidated) that is the detection of an error.
Codes have specific properties, which make them suitable for detection of specific error classes, such as single-bit errors, multi-bit errors, and burst errors.

Example:

1. Communication protocols (e.g. TCP/IP) and storage technology (e.g. RAID parity) are most prominent in employing coding checks.

Error *correcting* codes carry extra redundancy information to let the reader recover the original data when holding a data item that failed integrity verification. The amount of redundancy balances with the recovery strength of error correcting codes; some errors are too severe to be corrected by the code.

Error type: Data corruption

Benefits:

- The system processes (more) dependable, non-corrupted data
- Error detecting codes are freely available in (mature) libraries

Required system context:
The mechanism is only applicable to data[12].

[11] Error correcting codes only

[12] Note that an executable binary, stored in memory, is data.

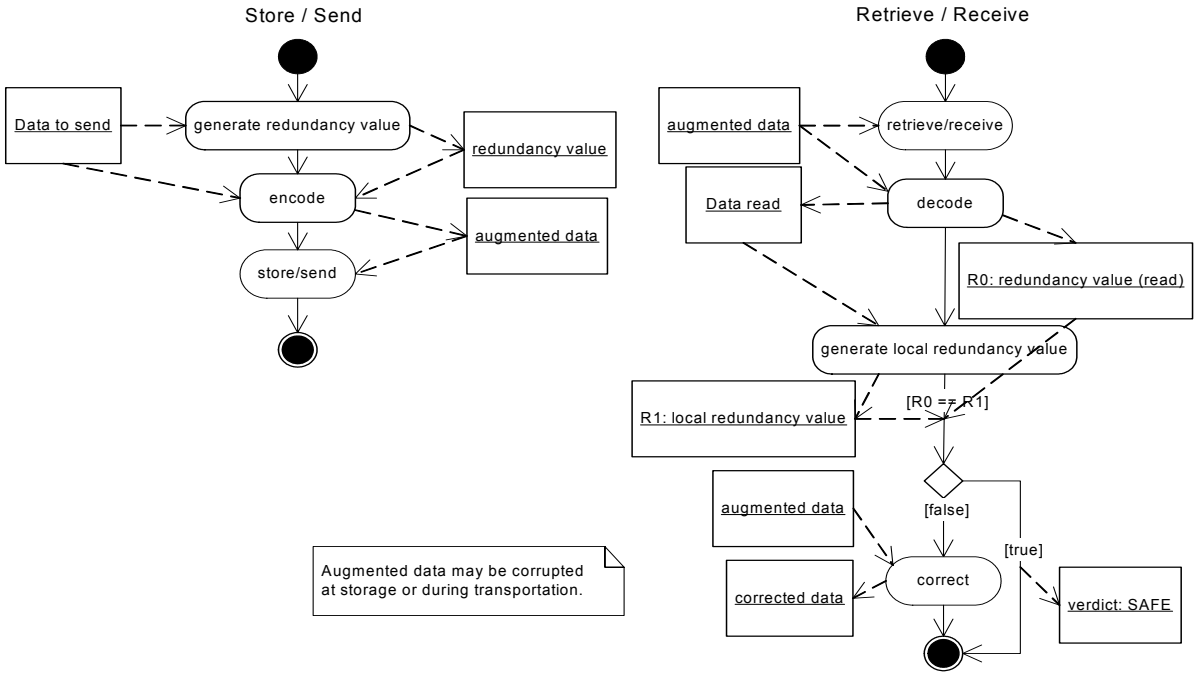


Figure 4: Error Detecting/Correcting Code, UML Activity diagram

Consequences:

Drawback: Reduced net bandwidth or reduced net storage capacity, due to the redundancy overhead.

Risk: Data marked as dependable based on its coding does not always convey dependable *information*: Successfully en-/decoded garbage remains garbage.

Risk: Corrupt data can pass the verification, rarely. No code is airtight. Each code is designed to detect a limited set of error classes.

Variations:

- By distributing data (and duplicates) over space or time, we can increase robustness against local or temporal errors (e.g. Compact Disc matrix).

Alternatives:

- Data Structure Integrity Check (Section 5.9)
- Reality Check (Section 5.5)

References:

- *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 (section 4.1.2 "Coding Checks")
- *"Error Detecting and Error Correcting Codes"*, R.W. Hamming, The Bell System Technical Journal, April 1950
- Applied in communication protocols, such as TCP/IP, for unreliable communication channels
- Applied for data storage on unreliable media, such as tape drives
- Reed Solomon encoding, used in Compact Disc and DVD
- Golay-, BCH- and Hamming Codes
- The software of a television is stored, CRC encoded, in the set's ROM. At runtime it is copied from ROM to RAM. Integrity of the RAM copy is CRC checked to ensure the copying process was dependable.

5.4. Resource Consumption Delimiter

Aliases: Memory Management Unit (MMU), resource expiration

Mechanism type: Error detection, Error containment

Description: Excessive resource requests are detected and contained.

Mechanics:
A Resource Manager manages the resources. The resource manager receives requests for resources –which it subsequently grants or denies– and enforces authorizations upon resource consumers. A violating request, including attempts at overconsumption, of a resource is detected as an error. (See Figure 5 and Figure 6.)
Authorizations can e.g. be expressed as maxima, ranges, a description of which parties can share access to a resource. Authorizations can also depend on the system state or the time. Consumption can e.g. be expressed as the amount of resources and the duration of mutual excluded possession of a resource.
Operating Systems provide resource managers for allocation of (among others) memory and semaphores. Resource Consumption Delimiters can be implemented through interface method checking when resources which are allocated via some interface (e.g. the semaphore and malloc interfaces). Note that if the resources (e.g. as in the case of memory and CPU cycles) can be consumed in a bypass of such interfaces, the Resource Manager must be implemented differently.

Examples:

1. An application requests more and more heap memory during execution, possibly due to some memory leak. When the heap memory manager detects that it allocated 2 MB of the heap to this application, it denies any further requests, to contain this error. The heap memory manager can warn the rest of the system.
2. Two processes, say *B* and *L*, compete for the CPU. *B* is allowed to consume "100ms of CPU time every 3 seconds", *L* can use what is left over ("best effort"). A CPU budget-scheduler will pre-empt process *L*, and replace it with *B*, when 2.9 seconds have passed since *L* started, to ensure that *B* is able to consume its budgeted CPU-time.

Error type: Resource overconsumption, memory leaks, unauthorized resource sharing

Benefits:

- Delimiters isolate resource consumers, containing them from each other's faults.
- Delimiters detect software errors, hence they rule out hardware as the fault that caused the error. This is useful for fault diagnosis.

Required system context:
Depending on the type of resource to protect, this may require additional hardware, e.g. a MMU is a hardware component required for memory protection. Operating system support (or functionality) may be required as well.

Consequences:
-

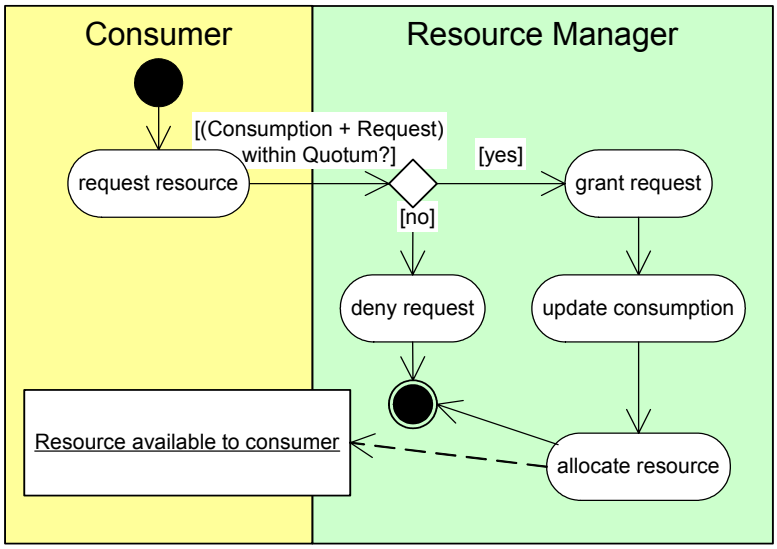


Figure 5: Resource Consumption Delimiter (request)

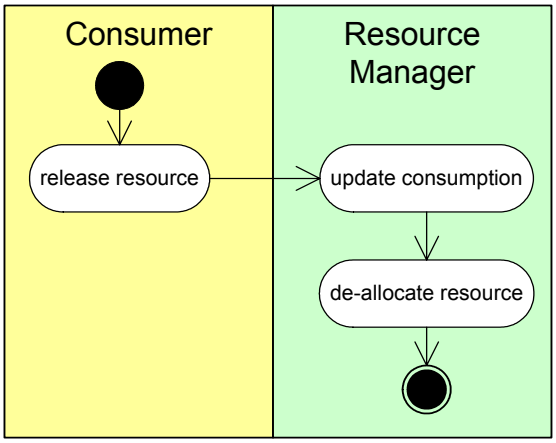


Figure 6: Resource Consumption Delimiter (release)

Variations:
The simplest resource consumption delimiter triggers an exception in the event of a resource overconsumption attempt. More advanced versions set resource quota per consumer. Even more advanced systems trigger an exception *before* the system runs out of resources for example by applying heuristics to monitored resource consumption over time to detect irregular consumption patterns.

Alternatives: -

References:

- Applied in the Memory Management Unit (MMU). This hardware component protects memory allocated to one process from being overwritten by any other process.
- Applied in the Hop Counter (Section 5.10)
- Applied in the Internet Firewall, wherein communicating partners are rejected access to specific TCP/IP ports.
- Provided by sandboxing, which delimits access to resources and system functions. E.g. JAVA applets cannot access the hard drive, due to the sandbox (the JAVA Virtual Machine).
- A CPU budget-scheduler allocates CPU time budgets to processes. See: Otero Perez, et. al, "*Resource management in multiprocessor systems*," in *Dynamic and robust streaming between connected CE-devices*, P. van der Stok, ISBN 1402034539, 2005.

5.5. Reality Check

Aliases: Common sense check, reasonability check

Mechanism type: Error detection

Description: Monitoring realism of input- and/or output values.

Mechanics:
An observer analyses the input into, output from, or observable state of a system (See Figure 7). The observer monitors all the data for *reality*, for example: if it falls within a pre-defined range, has a bounded rate of change, or follows predefined sequence. When observer reads an out-of-reality data item it signals an error.
The reality check can be based on interface specifications, domain models, physical limitations etc.
Observer can have a memory to store history, as for instance is required to monitor the *rate* of changes.

Example:

1. The realistic range can be dependent on the input, like to verify calculation of e^x we can use that for all x, $e^x \geq 1 + x$.

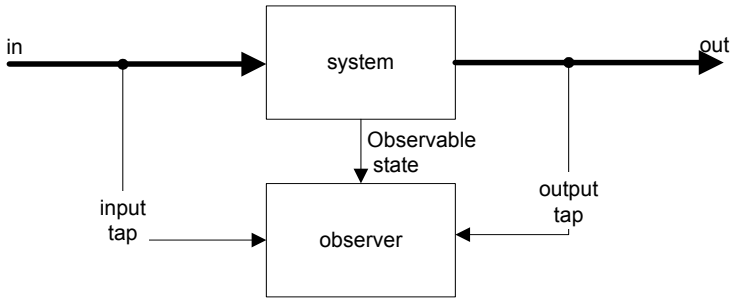


Figure 7: Reality Check

Error type: Violations of pre- and post-conditions, behavioural model errors.

Benefits: Unrealistic values are detected, which often means that the input is faulty, rather than the algorithm that is fed that input.

Required system context:
A model of realistic values or behaviour must be available for the system. For example a model for the input- and output ranges can come from a Requirements Document, stating "Person *age* must be a positive integer and the system will refuse values if unrealistic, i.e. *age* > 120."

Consequences:

Side effect: Updates to Requirements can require updates to realism checks (e.g. see [Ariane5]).

Risk: If the observer code is mixed-in with the system code, overzealous coding of assertions[13] can result in unreadable code.

Variations:

- Applicable to commands, data values, events, observable states, global variables.
- Observer can execute on the same thread as the system, enclosing the system code, or it can execute a-synchronously, independent of the system it observes.
- Observer can *partially* observe the system stream, to reduce overhead.
- Observer can monitor invariants that relate the output to the input.

Alternatives: Function Reversal (Section 5.8)

References:

- *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA / TM-2000-210616 (section 4.1.2 "Reasonableness checks")
- [Ariane5] http://en.wikipedia.org/wiki/Ariane_5_Flight_501

[13] Or any other form of checking assertions, e.g. adding "If(..)" or "ASSERT()" statements.

5.6. Self-test

Aliases: -

Mechanism type: Error detection

Description: Comparing the processed output of a injected (known) input with the before known result it should have.

Mechanics:
A tester injects a known input into the system's input stream and consumes the associated output from the system's output stream (see Figure 8). Tester *compares* the system's output with the expected result. If the actual result differs from the result the system is supposed to generate for this input, an error is detected. Meanwhile, the system consumes input values and produces output, agnostic of any tester's injections.

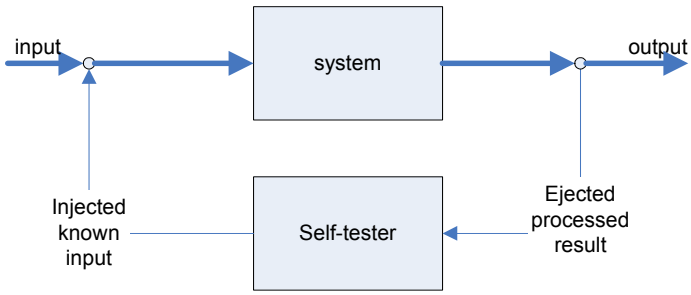


Figure 8: Self test

Error type: Errors affecting the output / result of the subsystem.

Benefits:
- Tests can be injected on convenient moments, e.g. when the system is idle.
- The timing of core functionality is not influenced.
- Tests can be constructed offline; hence design details of the system can be used. E.g. Unit tests can be re-used for this purpose.

Required system context:
- There must exist a verifiable relationship between input and output. For example *history effects* of the system may conflict with this requirement.
- The environment of the system must allow additional inputs to be injected and processed.

Consequences:

Side effect: This could introduce *jitter* in the system responsiveness. Test injections consume a time slot of the system.

Drawback: Tests are only injected *between* normal processing; hence no errors are detected *during* normal processing.

Risk: Previous results (before a detected error) could have been wrong. A solution could be to communicate this to the users of the results.

Variations:

- When to inject the test into the system can be decided by the environment (when it is possible, due to time consumption, or when an error is suspected) or periodically.
- Calibrate (Section 6.4), where the self-test is used to calibrate the system by adjusting settings.

Alternatives: -

References:

- This mechanism is used for physical systems or in chemical process industry, where system performance is tested against known inputs.

5.7. Compare Replicas

Aliases: Compare, Replication

Mechanism type: Error detection

Description: Detect errors by comparison; this can be of data or control.

Mechanics:
Replication checks make use of comparison to detect errors. This is a very general approach, which can be executed on *data* or on *implementations,* both in space and in time.
The assumption is that the objects compared should be identical, therefore if they differ, the mechanism detects that (at least) one of them is corrupt.

- In case of *data* comparison, replicas can be compared to detect data corruption.
- In case of *implementation* comparison, replicas of system output are compared. Many specific approaches exist to deliver such replicas, to name a few:
    - Run the algorithm two (or more) times on the same system and compare the results obtained after each run.
    - Develop multiple implementations of the same functionality, called Multi-Version Software (Section 8.4), and compare the results of each implementation's execution. Additionally, one can apply Voting (Section 6.2) to recover, which jointly is called *N-version programming*.
    - Run identical implementations on separate processors, and compare the results obtained at each processor. This is called Multiple hardware (Section 8.5).

Error type:
- Data corruption is detected by comparing replicas of the data stored on (e.g.) different storage media. This is replication in *space*.
- Implementation faults, detected if results of one implementation differ from the results of other implementations. This is replication in algorithm.
- Transient faults, detected when results of an algorithm vary between several executions of it (retry). This is replication in *time*.

Benefits: (negative) Trustworthiness: If the replicas differ, neither one is to be trusted.

Required system context:
When comparing replicas of data, they should be in comparable formats.

Consequences:

Drawback:	In order to supply the replicas extra resources are required. E.g. for *data* replicas this may be extra storage capacity. E.g. for *implementation* replicas this may be extra development effort, extra CPUs or consumption of more CPU cycles.
Risk:	Even if replicas are found to be identical, they may still be both/all wrong. If two replica's differ, either one (or even both) can be wrong.
Risk:	The benefit (negative trustworthiness) may be confusing: it cannot be conversed. If replicas are identical, this mechanism *cannot* deduce trustworthiness.

Variations:
Besides detection of errors, one can add recovery or masking by applying a vote as done for *implementation* replicas in Voting (Section 6.2) with Multi-Version Software (Section 8.4).

Alternatives:
- Error Detecting Code (Section 5.3) can be used instead of (large scale) *data* replicates, because they may require fewer additional storage resources.
- Function Reversal (Section 5.8). Inversing a function can be surprisingly attractive in particular cases.

References:
- *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 Chapter 4.1.2 " Replication checks", 4.1.5 "Process Pairs", 4.2.2 "N-Version Programming"
- Voting is applied in the space shuttle computer system (where five systems vote).
- Voting recovery is used in Philips Television software, where crucial data are saved three times in (physically) separate regions of FLASH storage.

5.8. Function Reversal

Aliases: Inverse operation

Mechanism type: Error detection

Description: Validate the result by applying the inverse operation and comparing the outcome with the original input.

Mechanics:
The output of a module is used as input for a reverse algorithm, this function outputs a computed input. This computed input is compared with the real input for validation of the output. (See Figure 9.)

Example:

1. Let *x* be the input value and *y* the output. One can verify that $y = x + 2$ by verifying the reverse function $y - 2$ equals the input, *x*.

Error type: Algorithmic implementation errors

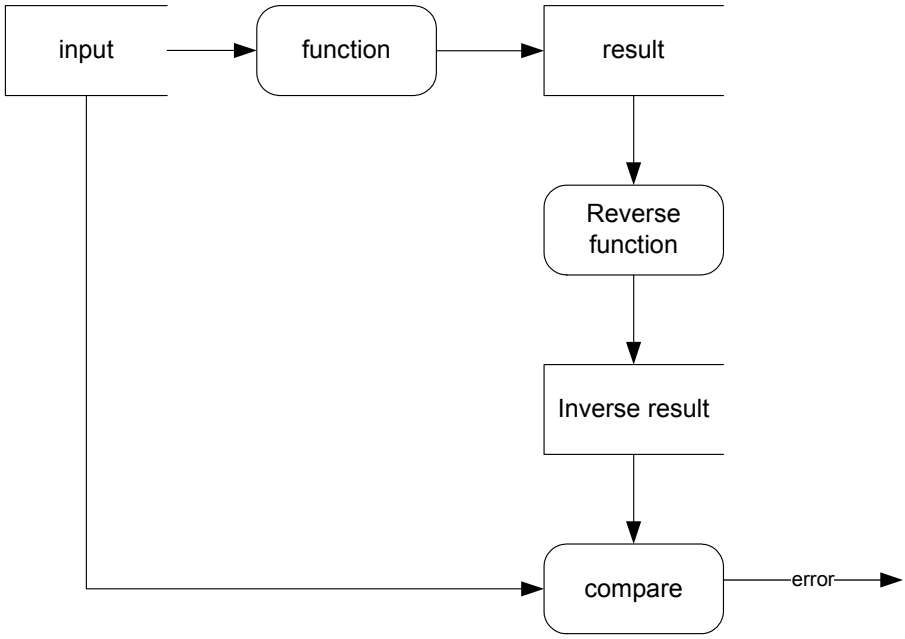


Figure 9 Dataflow function reversal

Benefits: Validated results.

Required system context:
An inverse algorithm must be available or possible. Some inverse functions have multiple correct values (possible inputs). For example, the result of y = x*x can be verified by checking whether x = sqrt(y) or x= -1 * sqrt (y).

Consequences:

Risk: The input data may be corrupt. However, one can apply a Reality Check (Section 5.5) to validate the input.

Risk: Due to rounding errors, correct results of floating point calculations may differ.

Variations: -

Alternatives:

- Reality Check (Section 5.5) whenever inverse algorithms are not available.
- Multi-Version Software (Section 8.4), wherein multiple implementations are used to verify calculations.

References:

- [NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 (4.1.2 "Reversal Checks")

5.9. Data Structure Integrity Check

Aliases: Canary

Mechanism type: Error detection

Description: Checking data-structure invariants

Mechanics:
A data structure check verifies that data structure invariants hold. This check can be run periodically. Typically, linked lists, queues, and trees have structural invariants. To facilitate this check, structures can be augmented with redundant data, such as (back-) pointers, counters, unique identifiers, and a canary.

Example:
1. [List] An invariant stating that field *counter* denotes the number of items in the list can be verified by counting the items contained in it.
2. [Tree] A *Balanced Leaf Tree* has -by design- depth *log(n)* and only its leafs may contain data. Both properties can be checked.
3. [Canary] A known value (the canary) is placed in (e.g. stack) memory around a buffer to monitor buffer overflow. When the buffer overflows, it will clobber the canary, making the overflow evident.

Error type: Data corruption, Memory writing errors

Benefits: The data structure can be assumed to be intact.

Required system context: The data structure must have verifiable invariants.

Consequences:

Risk: One may wrongly assume that if the structure is correct then all data inside it is correct as well.

Risk: Intelligent hostile stack smashes can copy the canary and remain undetected., see also [Smash].

Variations:
- Adding redundant data to reduce the overhead of the check
- Adding specific redundancy data might enable the data structure to recover from some error

Alternatives:
- Storing the data structure in *protected memory* prevents memory corruption caused by faults in peer processes. However, it does not protect against faults in code that updates/modifies the data structure.

References:

- *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 (4.1.2 "Structural Checks")
- [Smash] *http://Wikipidia.org* contains an article about employing the Canary mechanism to protect against stack smashing attacks, currently to be found at http://en.wikipedia.org/wiki/Stack-smashing_protection

5.10. Hop Counter

Aliases: -

Mechanism type: Error detection

Description: Limits the number of concurrent calls to a method.

Mechanics:
A Hop Counter limits the number of concurrent calls to a method (e.g. `foo()`) as follows:

- The Hop Counter is initially zero and incremented when the method is called.
- When the method returns, the Hop Counter is decremented.
- When the Hop Counter, i.e. the number of concurrent calls, exceeds the limit an error is signaled.

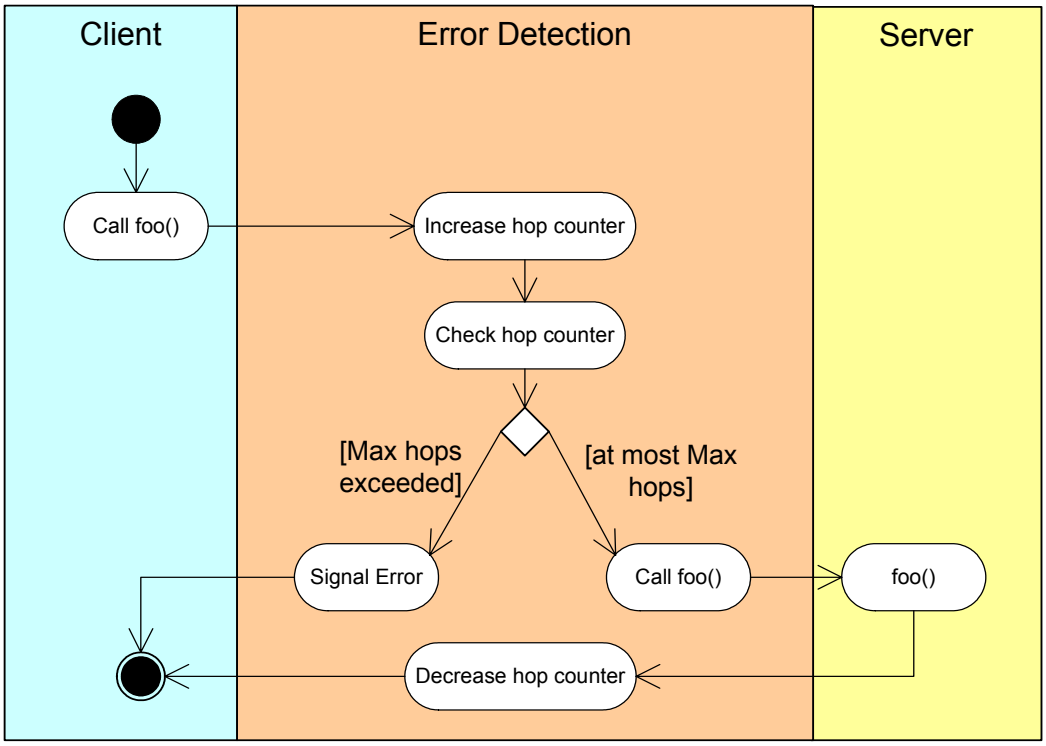


Figure 10: Hop Counter, UML Activity Diagram

It is up to the user of the Hop Counter how to recover from the *Signal Error*. Possible continuation strategies are:

- Call `foo()` anyway, while signaling an external error handler, without blocking the caller.
- Notifying the caller that the hop limit is reached. This implies that the mechanism is visible to the caller.

Error type:

- Infinite recursion, even over multiple threads, due to cycles in the call graph.
- Excessive number of concurrent calls to a method.

Benefits:

The Hop Counter can detect an excessive number of concurrent calls to a method, even from multiple threads. These excessive calls can be due to infinite recursion. These calls can result in excessive resource usage.

Required system context: -

Consequences:

Risk: A Hop Counter only works for explicit (external) calls to the method. It does not work for internal (implicit) calls to itself (real recursion).

Variations:

- The stop-condition of the Hop Counter (i.e. maximum number of concurrent calls attained) may be generalized e.g., "all available memory consumed" or "all available resources used".
- A Hop Counter can count access to a group of methods instead of to a single method.
- The Hop Counter can sample the thread identity of each call to `foo()`, and function as *call depth* counter.

Alternatives:

To prevent concurrent access to a method, one can use semaphores.

References:

- See http://www.hitech-projects.com/euprojects/space4u/. This pattern was demonstrated in the Space4U project.

5.11. Deadlock Detection

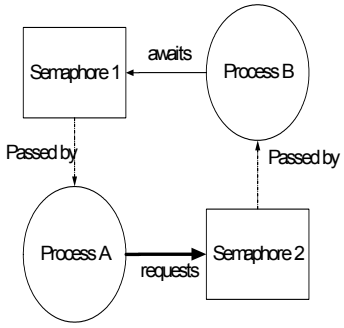
Aliases: Cyclic dependency

Mechanism type: Error detection

Description: Detects that processes have come into deadlock waiting for semaphores.

Mechanics:

A Sempahores Monitor wraps the semaphore API, provided by e.g. the OS. It intercepts all semaphore API calls of the Application to the Semaphore API provider, and tracks granted (locked) and requested (waiting) semaphores. The Semaphores Monitor searches for cyclic dependency in the current semaphore allocation. If it finds a cyclic dependency, the processes in the cycle are in deadlock.



Application creates and uses semaphores
Semaphores Monitor tracks all requested and locked semaphores, the requesting process and currently owning process, respectively.
Semaphore API provider, implements semaphore functionality. This is usually the Operating System.

A cycle of waiting processes, deadlock, can be detected instantly when it is constructed by a blocking request for a locked semaphore. In the figure above, *Semaphore 1* is requested and granted to *Process A*. Since then, *Process B* has been granted *Semaphore 2*. *Process B* has also requested *Semaphore 1*, and is currently blocked, awaiting its release. If now *Process A* requests *Semaphore 2*, it closes a dependency cycle with *Process B*, hence both processes can be detected to be in deadlock.

Error type: Deadlock in general, including deadlocks inside 3rd party binaries or originating from interaction therewith.

Benefits:

- Application agnostic: Does not require adaptations to the application's source code nor to its binary.
- Detects deadlock immediately, rather than by waiting for a timer.
- Quality assessment tool for 3rd party software; Internal (multi-threading) deadlock can be recognized, and fed back to the vendor. Liability is clearer too, since the mechanism can prove presence of deadlock in a 3rd party's binary object code.

Required system context: The semaphore implementation must allow interception with this detector.

Consequences:

Side effect: The deadlock detection algorithm incurs calculation overhead on each semaphore operation (for bookkeeping and analysis). Each semaphore acquire and -free operation will consume more CPU cycles.

Risk: None of the processes (in the interrelated set) may bypass the wrapped semaphore API , because that would allow some deadlocks to remain undetected.

Variations:

- Next to semaphores, other non-shareable, blocking resources can be introduced into the same cycle analyses.
- To reduce overhead, the most CPU consuming operation, cyclic dependency analysis, can be de-coupled into a separate thread, and run periodically, leaving only the administration to need updating upon every semaphore operation. Of course this reduces the immediateness benefit of this mechanism (see Benefits).
- One can add *recovery* to this mechanism, if it is possible to track all the resources that a process owns at any time, and to store that data when the process is killed, when attempting to resolve a deadlock. The process could then be re-started as soon as all resources it owned at the time of its demise are available again. (see Checkpoint and Restart, Section 7.1)
- To reduce overhead, the deadlock detection calculation can be deferred until CPU activity (of other processes) is low.
- Priority inversion can be a symptom of a partial deadlock. It may be possible to detect priority inversion by observing the processes respective priorities and CPU consumption.

Alternatives:

- Refusing applications' semaphore requests, if granting them would risk deadlock. Upon each request, this requires a calculation of complexity "NP-hard", as provided in *The Banker's algorithm,* Edsger W. Dijkstra, see [DELO].
- If source code is available, one can change the code to let all processes apply back-off, i.e. by letting them release all obtained semaphores upon encountering an unavailable semaphore (*2-phase locking*).
- Also, one can change the code to let all processes acquire semaphores following a partial ordering (*Ordered Semphore Usage*) [Arch].

References:

- Successfully applied in *REPA Demonstration of Deadlock Detection*, Felix Ogg, Philips Research 2005. (See Figure 11.)
- Order semaphore usage is used in television software, as described in [Arch] *Archnote 0011 Deadlock prevention*, Emiel Wijgerink, DSE-216032-EW-c7s3-0011, Philips Semiconductors, Eindhoven, 2002.
- [DELO] Wikipedia, section on Deadlock; http://en.wikipedia.org/wiki/Deadlock

- [LITTL] *The Little Book of Semaphores,* by Allen B. Downey, free PDF book from http://greenteapress.com/semaphores

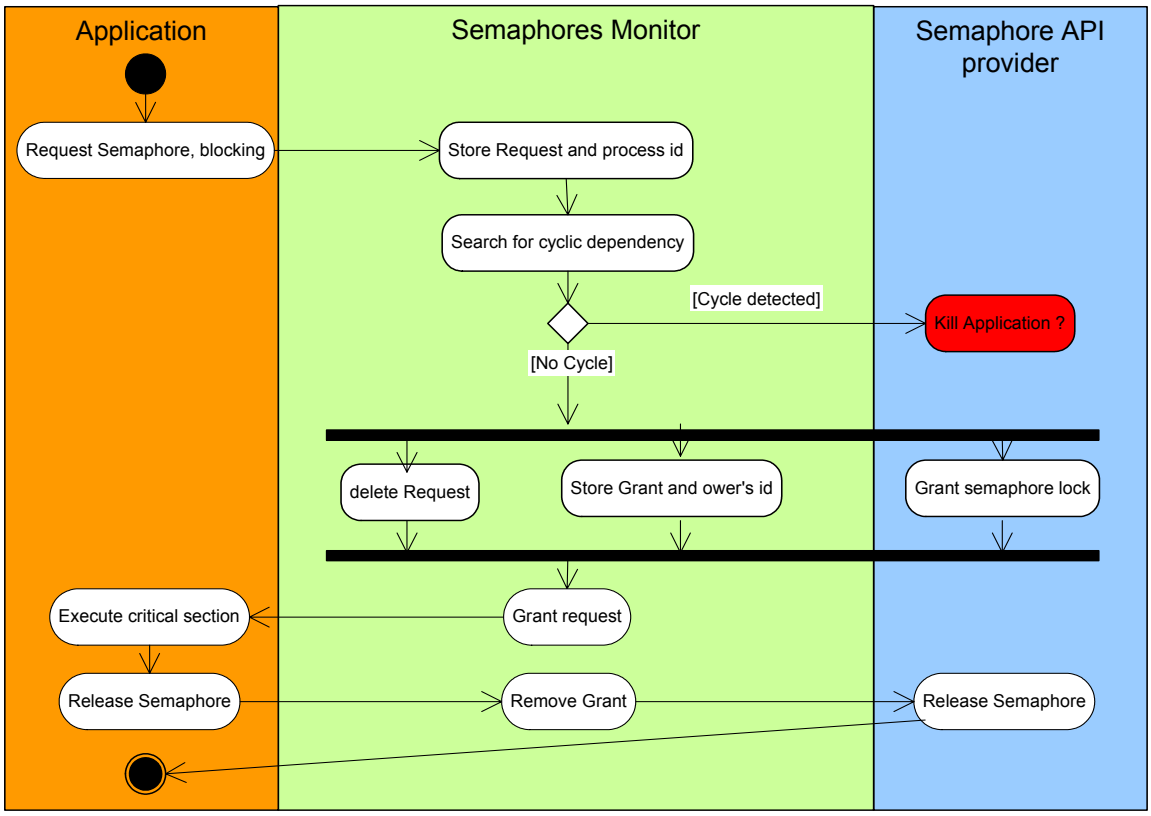


Figure 11: Run-time deadlock detection, REPA demonstration, UML activity diagram

6. Design Patterns for Error Recovery

6.1. Retry

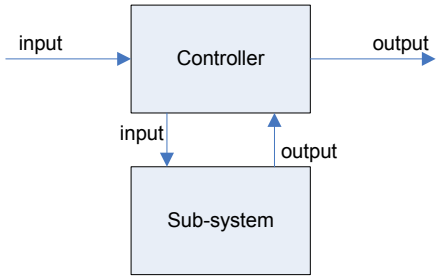
Aliases: -

Mechanism type: Error recovery

Description: Retry a function/operation after detection of an error.

Mechanics:
A controller retries a function, executed by a sub-system, if an error is detected. This "simple" retry mechanism is based on Wrapping (Section 8.1). The wrapper provides the ability to retry without changing the environment of the sub-system.

How to detect the error is in principle independent from this retry mechanism. Retries can be done when, for example, error codes or exceptions are returned by the sub-system. The sub-system could indicate that resources are unavailable.



Error type: Transient faults, like temporarily unavailable resources.

Benefits:
- Overcomes transient errors.
- Users are agnostic to temporary errors in the sub-systems.

Required system context:
- It must be possible to just retry this function, i.e.:
  - Possible (external) side effects must be allowed to repeat too
  - The function is state-less.
- The retried function should always terminate (in some way), but e.g. Time-out (Section 5.1) can be used to artificially terminate a non-terminating function.

Consequences:

Side effect: Retrying takes time, which might be unacceptable.

Drawback: Retries are only possible for state-less sub-systems. See alternatives for solutions.

Drawback: Need to design a stop-criterion, to prevent endless retries.

Risk: The controller keeps retrying infinitely (endless retries, life-lock).

Risk: Retry only recovers *transient* faults, because the retried function is exactly the same. Retries cannot recover *persistent* errors. This leaves the question of how to diagnose that an error is transient rather than persistent?

Variations:

- A stop-criterion is used to prevent endless-retries. For example, the maximum number of retries or the time to spend retrying.
- Random waiting time before retries can be applied, for example to prevent collisions between multiple parties retrying at the same time. As is used in Internet protocols, like TCP/IP.

Alternatives:

- If functions are not state-less, the Checkpoint and Restart (Section 7.1) allows functions that have internal state to restart / retry.
- Recovery Block (Section 7.2), which combines Multi-Version Software (Section 8.4) and Checkpoint and Restart (Section 7.1), can also cope with errors in the implementation.
- Multiple hardware (Section 8.5) can be used to retry on a different processor, which can also cope with hardware defects or design flaws.

References:

- "Method Retry" (see Figure 12) implemented in the Space4U project is an instance of this pattern.

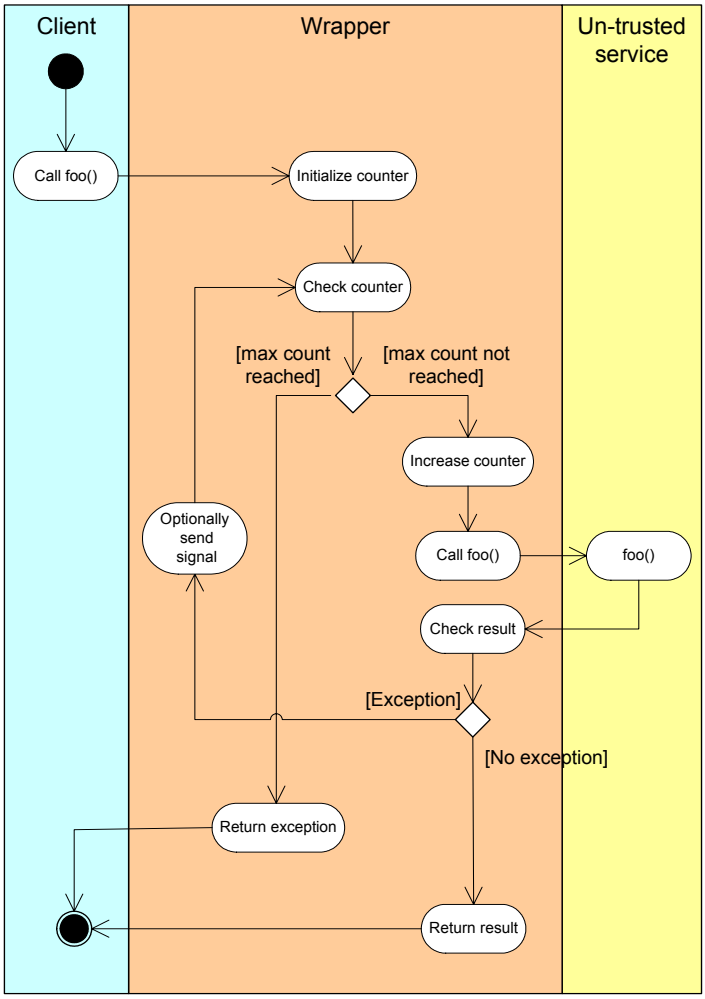


Figure 12: Method Retry, Space4U, UML activity diagram (adapted for this report)

6.2. Voting

Aliases: Voting unit, N-version programming

Mechanism type: Error masking

Description: Combining multiple results into a single, more reliable result.

Mechanics:
The input is fed to multiple embodiments (e.g. implementations) of the same functionality. Each embodiment delivers a result. A voting unit selects one of the available results based on a voting scheme (e.g. Majority voting, see variations).
If the outcome of the vote is inconclusive an exception is raised, otherwise the selected result is promoted to final result.

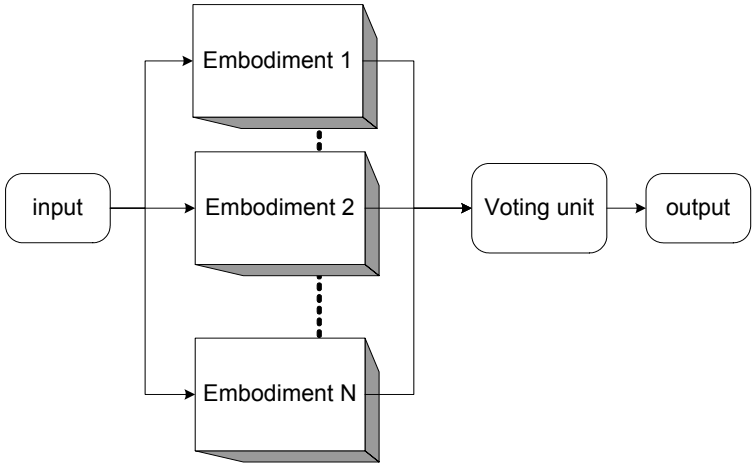


Figure 13:Voting

Error type:
- All errors in the results of the embodiments.
- Algorithmic implementation errors, causing wrong results (because detection is based on comparing the algorithmic results).

Benefits: Reduces the probability that an error in one of the embodiments leads to a failure.

Required system context:
The system must allow for resources (e.g. time!) consumption of running all the alternatives and the voting unit

Consequences:

Risk: The assumption is that the majority is right. Sometimes the correct result is only found by a minority, and consequently overruled, leading to an incorrect final result.

Risk: In some voting schemes a vote may be inconclusive. Additional error handling is still needed.

Risk: The voting unit is a single point-of-failure. Redundant voting units reduce/remove this risk.

Variations:

- "N-version programming" uses Multi-Version Software (Section 8.4) to implement the embodiments.

Variations to the voting scheme:

- Unanimous[14] vote, all votes are equal
- Majority Voting (>50%), or any other fraction
- Maximum Likelihood Voting weighs each of the embodiments' results in the final result. It is never inconclusive.
- Median Voting always delivers a non-extreme answer. It is never inconclusive.
- Only include votes that were delivered before a deadline, using Time-out (Section 5.1).
- At arrival of each vote, the voting process may be concluded if sufficient votes are in. For example, when two of three votes are in and identical, the outcome of Majority Voting is decided already.

Alternatives: Recovery Block (Section 7.2)

References:

- [Avizienis] Algirdas Avizienis, *"The Methodology of N-Version Programming"*, in M. Lyu editor, *"Software Fault Tolerance"*, John Wiley & Sons, 1995.
- [NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616
- [Avi1977] A. Avizienis and L. Chen. *"On the implementation of N-version programming for software fault tolerance during execution"*, In Proc. IEEE COMPSAC 77, pages 149-155, November 1977.

[14] In Fault Tolerance literature this is termed *Consensus Voting.*

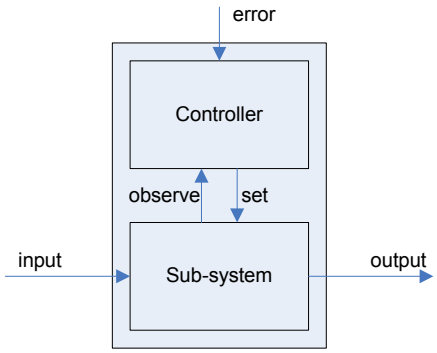
6.3. Roll-back & Roll-forward

Aliases: Reset, re-boot, fail state, safe state, re-initialize

Mechanism type: Error recovery

Description: A sub-system is set to a known state.

Mechanics:
A controller *sets* the sub-system to a known valid or safe state. In case of rollback this is a previous state and for roll-forward this is a next state. A well-known state to go to is the initial state, which often is achieved by a restart or reboot. Context / state information can be taken into account to determine to which state to go, in which case the controller must be able to *observe* the current state. This controller could be build into the sub-system.



The environment of the subsystems needs to be *synchronized* (or notified) if a sub-system goes to a new state autonomously. One way to achieve this is to model this behavior into the sub-system. The safe states that are used to go to because of a rollback or roll-forward can be designed to take care of this resynchronization with the environment.

Error type:
Any error that brings the sub-system in an unwanted state, causing illegal state transitions.

Benefits:
Sub-system is in a stable state after the roll-back/-forward.

Required system context:

- It must be possible for the observer to *set* the subsystem to another state and incase needed to *observe* the current state.
- It must be possible for the system to continue after the state change of the sub-system. There are no external side effects that should not be repeated in case of a rollback, and no external side effects that might be skipped in case of roll-forward.
- Additional synchronization might be needed between the sub-system and its environment. This might imply that state changes due to roll-back/-forward are anticipated and designed in.

Consequences:

Drawback: Synchronization with the environment might be needed after a state change.

Risk: Moving to another state might cause side effects to the environment or synchronization problems.

Variations:

- State information can be used to determine the state to go to.
- The controller can be embedded into the sub-system, in which case there still needs to be a way to tell the sub-system that an error has occurred.
- Checkpoint (Section 8.3) can be used to save state information.
- Checkpoint and Restart (Section 7.1) is actually a retry mechanism (but retrying is a special case of rollback/-forward).

Alternatives:

Retries (with checkpoints or journaling) might eliminate the need for synchronization, because the sub-system will eventually be in the desired state.

References: -

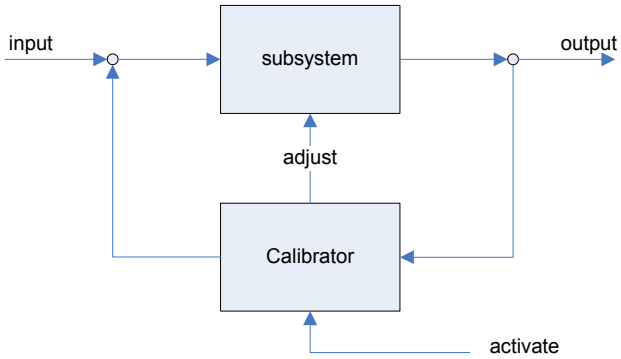
6.4. Calibrate

Aliases: -

Mechanism type: Error detection and error recovery

Description: Adjust the system based on measurements done by self-testing.

Mechanics:
A Self-test (Section 5.6) is combined with a recovery mechanism that repairs the tested system if the test fails.
A calibrator injects a reference input into the system and *compares* the processed output with the expected (known) result to detect any errors. Upon an error, the calibrator adjusts the subsystem, in a continuous feedback-loop.



Few "software-only" systems require calibration. Calibration is applicable to systems interacting with the physical world, such as robotics and hardware controllers.

Error type:
- Errors affecting the output (stream) of the subsystem
- Accumulating rounding errors in floating point calculation

Benefits:
- Repair of functionality (instead of only masking).
- Control drifting systems, these are most often processing external input with variations.

Required system context:
- A verifiable relationship between input and output must exist (see Self-test, Section 5.6)
- The environment of the system must allow additional inputs to be injected and processed.
- It must be possible to adjust the system, dynamically.

Consequences:

Side effect: Calibrating takes time, and the calibration algorithm may be unpredictable. This can cause *jitter* in the process flow through the system.

Risk: Rebooting (part of) the system may offer faster recovery.

Variations:

- The adjustment can consist of a code image overwrite, i.e. flashing the ROM.
- Activation can be periodical or triggered by error detection elsewhere in the system.
- Instead of "binary" calibration, a bounded range calibration can accept deviating results within limits.
- Calibration can be built into the system. The system can supply an external calibration trigger interface to its environment.

Alternatives: -

References:

- Calibration is used in chemical process industry
- Recent embedded software in the DVD/Blu-Ray recorder contains software driven calibration of the reading heads and the laser.

6.5. Drill Sergeant

Aliases: Periodic refresh, Repeated command, repeat state

Mechanism type: Error recovery

Description: Commands are repeated.

Mechanics:
A Repeater repeats commands from the Commander to the Slave-subsystem periodically. This mechanism is designed for systems where the commander cannot trust the slave-subsystem to stay in the right *state*, for example when hardware tends to lose settings. This only works if there is some kind of Master-Slave relation between the commander and the slave-subsystem. This is often the case in control systems.
This mechanism is for example used in televisions to control tuners that sometimes lose their frequency setting, probably due to electromagnetic interference. The repeater (over)writes the (write only)registers periodically with the same value.

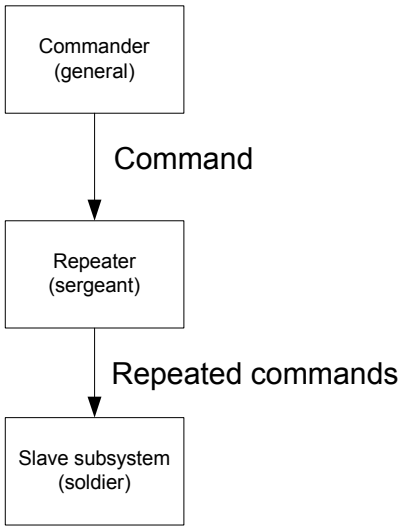


Figure 14: Drill Sergeant

Error type:
- Loss of state by the slave-subsystem, for example caused by the environment (Electromagnetic radiation).
- Drifting state of the slave (e.g. numerical instability)

Benefits:
- The slave-subsystem is (re-)set into the desired state.
- The time a sub-system can be in an "undesired" state depends on the Repeater period. The period must be chosen such that the undesired state only causes acceptable artifacts. (Risk management). The worst-case situation lasts at most the repeat period of time.

Required system context:
There must be some Control or Master-Slave relation between the two subsystems.
The repeating of the command should not cause side effects.

Consequences:

Side effect: Periodic communication.

Risk: The mechanism might become unnecessary when the risk becomes too low (the error does not happen anymore, due to a redesign). The problem might happen too seldom for the mechanism to be profitably implemented.

Drawback: Not all commands can be repeated. Repeating commands like “increment” can cause unwanted side effects. Typically, commands that contain some notion of state can be repeated, like “Set”-commands.

Variations:
- The hierarchical roles of the Commander and the Repeater can be merged into a single actor.
- The slave-subsystem can poll the state regularly. This only works for active slaves, not for hardware registers. This mechanism might be called "The uncertain soldier" or "Am I doing it right?”

Alternatives: -

References:
- Used by parent to keep their children from forgetting what they were told due to distractions.
- Used in televisions as shown in the example described in the mechanism section.

7. Composite Design Patterns

In the preceding chapter we presented several atomic mechanisms to detect and recover from run-time software errors. In this chapter we present composite mechanisms that consist of multiple atomic mechanisms. A typical example would be to combine one or more detection mechanisms with a recovery mechanism.

Again, this inventory is by definition incomplete. This chapter should be regarded as a demonstration of mechanism compositions. The writers hope to inspire the reader to make new combinations to suit the problems which he or she faces.

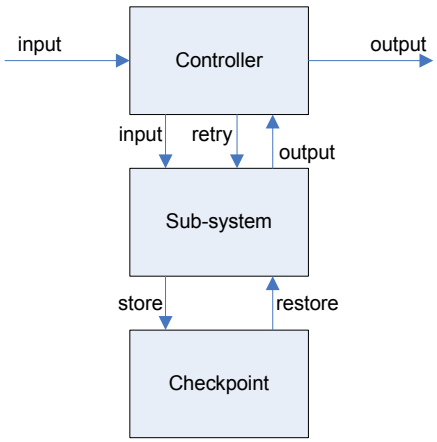
7.1. Checkpoint and Restart

Aliases: Retry with state

Mechanism type: Error Recovery

Description: Retry saving the start state so that each retry has the same context.

Mechanics:
The Controller performs Retry (Section 6.1) based on Wrapping (Section 8.1) which isolates the sub-system from its environment. The controller determines when to retry (based on error detection). The sub-system always *stores* its state in a Checkpoint (Section 8.3) before an input is processed. If the controller detects an error it signals the sub-system to *retry*, after which the sub-system will *restore* the state from the checkpoint before it runs again.



Error type: Transient errors, temporarily unavailable shared resources.

Benefits:
- Overcomes transient errors.
- Users are unaware of temporary errors in the sub-systems.
- Enables advanced (with state) retries, in contrast with the "simple" retry, which is state-less.

Required system context:
- It must be possible to just retry this function; there are no existing unrecoverable actions. (You cannot launch a spaceship twice.)
- It should be determined how to prevent endless retries / life-lock. For example: the maximum number of retries or the maximum time spent retrying.
- The retried function should always terminate (in some way).

Consequences:

Drawback: Retries are only done if errors are detected, in other cases errors might propagate.

Side effect: State is saved before every action.

Risk: Life-lock, the controller keeps retrying the sub-system infinitely.

Risk: The checkpoint may save an error state.

Variations:

- An external error handler can be added to handle the case that the maximum number of retries is exceeded
- Random waiting time before retries can be applied, for example to prevent collisions between multiple parties retrying at the same time.

Alternatives:

- Roll-back & Roll-forward (Section 6.3) jumps to static (defined at compile time) states, instead of to any dynamically (run-time) stored state.

References:

- [NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 Chapter 4.1.4 and Chapter 4.1.6/7.

7.2. Recovery Block

Aliases: dynamic redundancy, standby spare[15].

Mechanism type: Error recovery, multi-version software

Description:
Retry multiple implementations sequentially whereby the same start state is fed to each retry.

Mechanics:
This multi-version programming retry uses alternative implementations of the functionality. After running an alternative, the result of the alternative is tested with an acceptance test. If the acceptance test fails another alternative is tried. Each alternative needs the same start situation and thus a checkpoint mechanism is used to provide this.

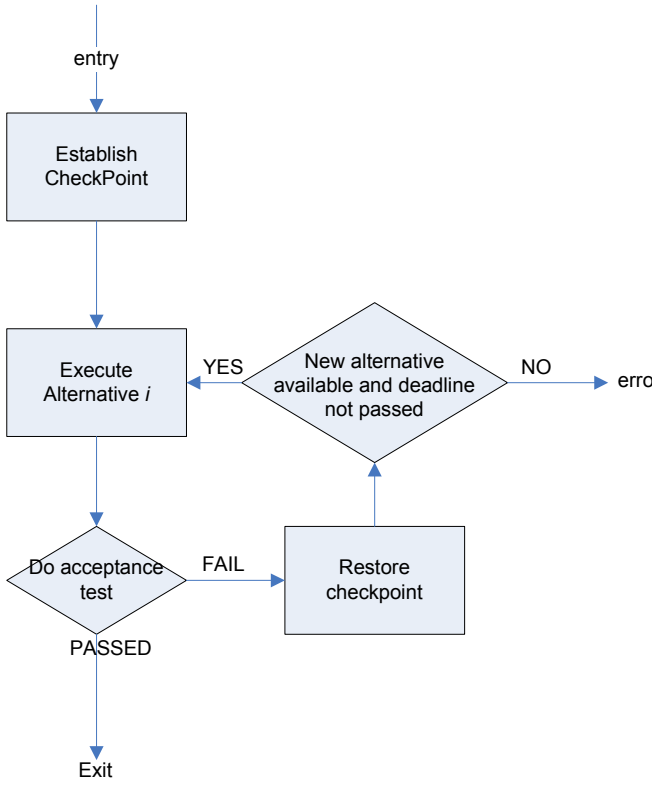
Error type:
Algorithmic implementation errors, causing wrong (not accepted) results

Benefits:

- The state is preserved if something goes wrong.
- An exception is raised when a final error is detected.
- Retries are only done when the acceptance test fails.

Required system context:

- Acceptance test criteria



[15] in hardware

Consequences:

Side effect: In case of failures, more time is needed to run alternative versions

Side effect: Requirement changes affect all N versions.

Drawback: Multiple alternatives need to be implemented

Risk: Only faults detectable by the acceptance test can be retried, which might give a false sense of security.

Variations:

- Variations to the acceptance test. Some known acceptance tests are:
  - Range checking, a Reality Check (Section 5.5)
  - Compare with reverse operation, Function Reversal (Section 5.8)
- Compare with the result of the next alternative. This is called "Recovery Block Version" (RBN), which uses *compare* as acceptance test and needs the result of two alternatives *n* and *n+1*. This is similar to "N-Version Programming" which combines Multi-Version Software (Section 8.4) with Voting (Section 6.2).
- Create alternatives by using different tool chains or compiler optimization settings, to create different binaries to overcome bugs in the tool chain.

Alternatives:

- A simple Retry, without alternatives.
- A Rollback with some test criteria to detect errors.
- Conversation [RAND75] is "*a concurrent extension of the Recovery Block scheme: A checkpoint is taken when each process enters a Conversation. This process may communicate only with processes already entered in the Conversation to ensure that no information smuggling (that is, spreading of not yet validated data) can take place. When all processes have completed their operations, a global acceptance test takes place; in case of failure, all processes roll back and execute their next alternatives. If the acceptance test is satisfied, all processes may discard their checkpoints and proceed, exiting synchronously from the Conversation.*"[16]

References:

- [RAND75] B. Randell, "*System structure for software fault tolerance*" IEEE Trans. Software Eng., Vol.Se-1, pp.220-232, 1975.
- [RAND95] [Lyu95] B. Randell and J. Xu, *"The Evolution of the Recovery Block Concept"*, in M. Lyu editor, *"Software Fault Tolerance"*, p1-21,, Wiley, 1995.

[16] Quoted from: Clematis and Gianuzzi - A Hierarchical structure for fault tolerant reactive programs - ACM-SAC 1993, pp. 208-214.

- [Avizienis][Lyu95] Algirdas Avizienis, *"The Methodology of N-Version Programming"*, in M. Lyu editor, *"Software Fault Tolerance"*, John Wiley & Sons, 1995.
- [Horn1974] J.J Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randall, *"A program structure for error detection and recovery"*, Lecture Notes in Computer Science, vol.16 pp.177-193, 1974.
- Recovery Block is similar to the dynamic redundancy (standby spare) technique in hardware.
- Applied in Aerospace and Nuclear plants since the 1970s.

8. Supporting patterns

In this section we describe technology that facilitates detection and recovery mechanisms or combinations thereof. This technology does not add fault tolerance to the system per se, but instead it *enables* (other) fault tolerance mechanisms to function.

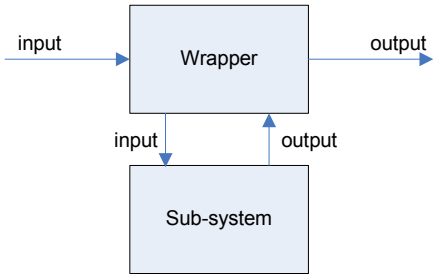
8.1. Wrapping

Aliases: encapsulation, isolation, and containment

Mechanism type: Error containment, (a detection and recovery facility).

Description:
The sub-system is hidden within a wrapper that shields the environment from potential errors in the sub-system.

Mechanics:
The wrapper handles the interaction from the environment with the sub-system. The wrapper can contain code that provides checks and recovery mechanisms (like pre- and post-condition checking or retry mechanisms).



Error type: Error containment

Benefits:
The wrapper provides ***isolation*** of the sub-system from the rest of the system and the other way around. The wrapper enables ***containment*** for all errors in the sub-system that it can handle.

Required system context:
All interaction of the sub-system needs to be known. Explicit modeling techniques like component-based techniques are very beneficial.

Consequences:

Side effect:	Indirections in the calls.
Drawback:	Calls to generic / infrastructure facilities that are not wrapped are not isolated.
Drawback:	Some behavior cannot be wrapped, for example, CPU cycle usage and direct (memory mapped) IO[17]
Risk:	Indirections may degrade system performance, both due to extra code, and due to blocking of compiler optimizations.

Variations: -

Alternatives:

- An alternative for isolating CPU cycle usage is to use an operating system that allows cycle budgets.
- Sandboxing, e.g. by running each sub-system on a separate Virtual Machine. Products like VMWare and Xen support this. Also, in JAVA, by running a separate JAVA Virtual Machine for each sub-system. Note that this does not prevent resource problems, but delegates such problems to the provider of the Virtual Machines.

References: -

[17] which should be accessed through a device driver, which is a wrapper in itself .

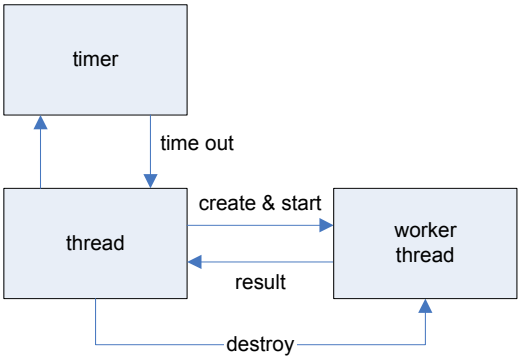
8.2. Worker thread

Aliases: Asynchronous function call

Mechanism type: Error containment

Description: Executing part of the system in a separate thread, which is isolated from the rest of the system, which executes in the master thread.

Mechanism:



A worker thread can be used for *error containment*: When a master thread must execute some functionality, it creates a new thread, called the worker thread, to run this functionality. When this worker thread completes its work, the result is communicated back to the master thread and the worker thread is destroyed. See also Figure 15.
If, the worker thread fails – for any reason - to provide the result, the master thread can recover, since it is *independent* of the worker thread. See also Figure 16. The failing of the worker thread can be detected by a time-out of a timer.

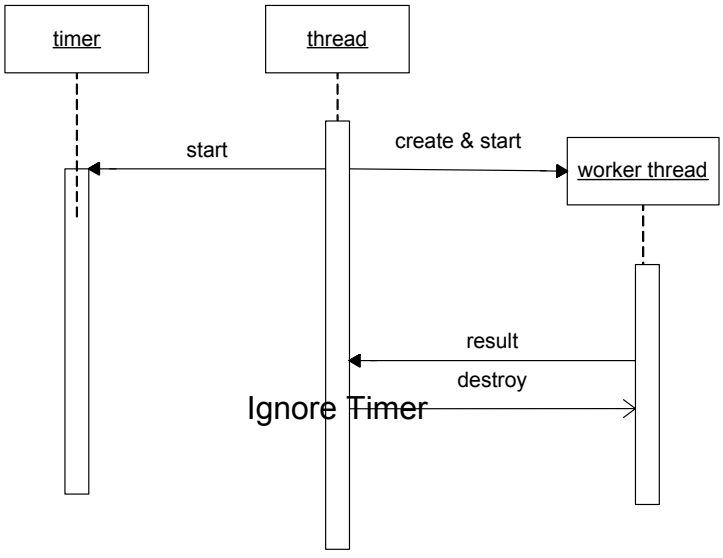


Figure 15 Succeeding worker thread.

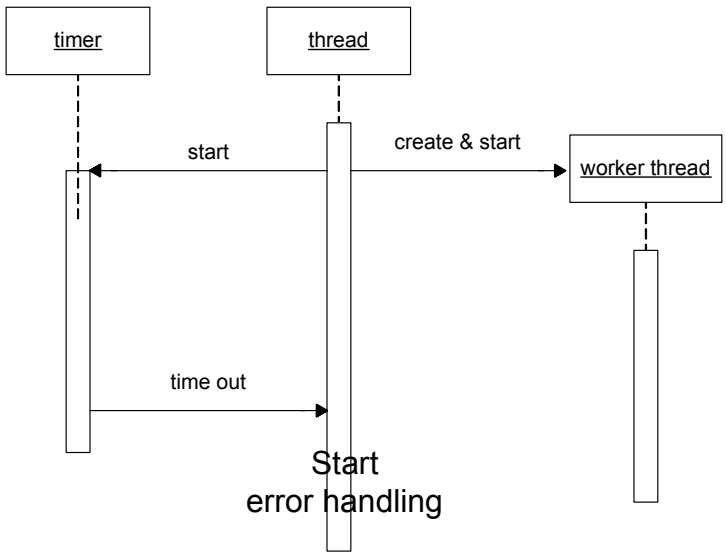


Figure 16: Failing worker thread.

Error type: lock-ups, too late responses, life-locks.

Benefits: The error is contained in the newly created thread, which enables the master thread to recover.

Required system context: The system must support multi-threading.

Consequences:

Drawback: Creating and destroying threads is time-consuming.

Drawback: Inter-thread function calls impose more overhead than direct (i.e. intra-thread) function calls.

Variations:

- A worker *process* instead of a worker *thread* can be used. Process separation provides additional isolation at the cost of additional overhead. See [Wikipedia] for a detailed description of the difference between processes and threads.
- To reduce the overhead of thread creation and destruction, one can employ *thread pooling*. However, if failing threads remain in the thread pool, errors can propagate.

Alternatives:

- Functionality can be executed on the main thread, synchronously.
- A checkpoint can be made before the function is called. If the function fails, the system can roll back to that checkpoint.

References:

- http://www.flounder.com/workerthreads.htm

- [Wikipedia] http://en.wikipedia.org/wiki/Thread_(computer_science)

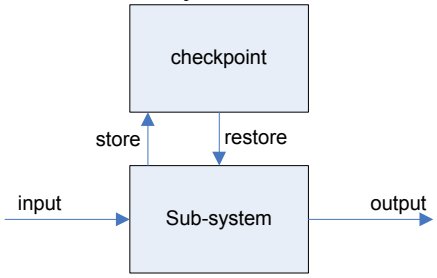
8.3. Checkpoint

Aliases: Remember correct state

Mechanism type: facility

Description: Saving the system state externally.

Mechanics:
The internal state of a sub-system is saved (*stored*), in the (external) checkpoint, at defined points. These points can be at fixed intervals (time) or fixed places during execution. The Checkpoint should contain a safe state. The Sub-system is able to go back to this state, *restore*, and continue correctly afterwards.



Error type: a facilitating mechanism for recovery strategies

Benefits:
The state of a sub-system is remembered externally and thus protected from possible problems in the sub-system.
Provides a means for retry for sub-system that are not stateless. Checkpoints might be used to rollback to, for example as used in Checkpoint and Restart (Section 7.1) and in Recovery Block (Section 7.2).

Required system context:
There must be a set of variables that can define the state of the sub-system in all cases.
The sub-system is able to go back to a check-pointed state, after an error, and operate correctly afterwards.

Consequences:

Risk: The saved state might not be safe. This can, for example, happen with periodic check pointing, if the time to detect an error is longer than the save period.

Variations:

- Reset the sub-system (instead of restore the state) to eliminate possible causes not recovered by the restore (like corrupted code).
- Combine with journaling (Section 8.6), if check pointing is not done at every state change. The combination with journaling might make it possible to go back to the exact state before the sub-system was "interrupted". This might be more efficient if the size of the state is big compared with the size of the inputs.
- The saved state might be stored as a dynamic structure. The sub-system knows what to save in which cases and how to reproduce this when reading back the checkpoint information. This requires careful design of the state information to be saved to safeguard reproducibility.
- The operating system may provide a generic checkpointing feature, saving the entire memory space (in use by the sub-system) as a checkpoint. Such a feature is independent of the sub-system.

Alternatives: Journaling (Section 8.6)

References: -

8.4. Multi-Version Software

Aliases: Algorithmic redundancy

Mechanism type: Error detection, error containment[18]

Description: Software that contains multiple, unique implementations of the same functionality.

Mechanism:
Multi-version software compares the results of executing multiple implementations of the same functionality. When the results differ an error is signaled.

Examples:

1. Two programmers each implement the same, complex algorithm. Both implementations are run. If their results differ, either implementation is wrong.

Error type: Errors caused by design faults, e.g. implementation errors, errors resulting from incomplete specifications

Benefits: Multi-version software detects design faults, which are typically difficult to find.

Required system context: Must allow for time to run and compare all the versions implemented.

Consequences:

Side Effect: Developing and running multiple implementations of the same algorithm multiplies the development and/or hardware costs.

Drawback: To be able to compare the results of the multiple implementations of the same algorithm, they all should have the same input. Therefore, additional synchronization is needed for streaming input data.

Risk: Errors resulting from a fault in the Requirements Specification go unnoticed when all implementations are based on that same specification. (Common Mode Failure)

[18] for the Voting variation

Variations:

- The implementations can be run in parallel or in sequence. Furthermore, the implementations could be used for testing purposes only, to reduce resource consumption in the final product.
- Unique implementations can also be generated binaries from different compilers (such as GNU GCC and Borland C), generated from the same C sources.
- Also, Voting (Section 6.2) can be applied to recover from the situation wherein one result deviates from the others. This is called *N-version programming* and can be considered error containment.

Alternatives: -

References:

- [Avizienis][Lyu95] Algirdas Avizienis, *"The Methodology of N-Version Programming"*, in M. Lyu editor, *"Software Fault Tolerance"*, John Wiley & Sons, 1995.
- [NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616
- [Avi1977] A. Avizienis and L. Chen. *"On the implementation of N-version programming for software fault tolerance during execution"*, In Proc. IEEE COMPSAC 77, pages 149-155, November 1977.

8.5. Multiple hardware

Aliases: -

Mechanism type: Error detection, error recovery

Description: Sub-system can simultaneously be executed on multiple hardware instances (multi-processor).

Mechanics:
Multiple hardware (multi-processor) is used to create different executions of a sub-system. Hardware defects (broken) of the processor can be exposed when using the same type of processor; design faults of the processor can be exposed when different types of processors are used.
Different executions can be used with *retry* and *compare* approaches.

In case of *retry*, executions are run sequentially. Each executing can be run on different processors. Different variants of retry might be chosen, based on their properties. These can be single version variants like Retry (Section 6.1) or Checkpoint and Restart (Section 7.1), or the multi-version variant as described in Recovery Block (Section 7.2).

In case of *compare*, executions are run in parallel. Each execution can be run on a different processor. Different variants of compare might be chosen, based on their properties. This can be the Compare Replicas (Section 5.7), or Voting (Section 6.2), which can mask errors. Different processors might also be deployed in the multi version software variant of voting: "N-Version Programming". Note that Compare Replicas and Voting run on a single processor, and thus become a single point of failure, which needs to be addressed in the overall design.

Error type:
- Hardware errors
- Defects (broken hardware): can be addressed with more processors of the same type
- Design faults (like the Pentium bug): can be addressed with different types of processors.
- Tool chain errors
- Different processors need different tool chains (compilers, linkers, assembler, loader & libraries), which might have different faults.

Benefits: Hardware errors can be addressed.

Required system context:
The system must contain more than one processor to run the software.
In case of *retries* there should be no unrecoverable actions (like a missile is fired).

Consequences:

Drawback: Redundant hardware is not always available.

Risk: (In case of Retry) The error caused by the hardware must be caught by the error detection before this mechanism will be useful. The quality of this solution thus depends on the detection mechanism used.

Risk: (In case of “Compare replicas”) If the comparison is run on a single processor it can be a single point of failure.

Variations:

- In the literature we found "Process Pairs" which sets a checkpoint and restarts on another CPU.
- Multi-version programming variants can run on multiple processors, for example multi-version software combined with a Voting on the multiple results. This is called "N-Version Programming".

Alternatives:

- Retry (Section 6.1) or Checkpoint and Restart (Section 7.1) can recover transient errors.
- Compare Replicas (Section 5.7) may provide similar error detection.
- Multi-version programming, like in Multi-Version Software (Section 8.4)

References:

- [NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616 (4.1.5 "Process Pairs")

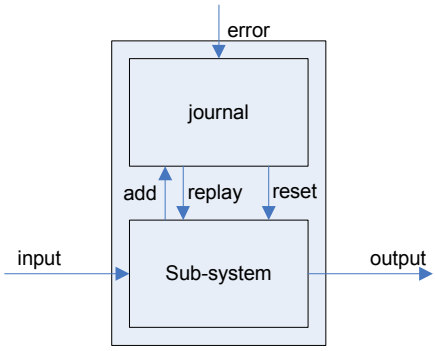
8.6. Journaling

Aliases: replay, history

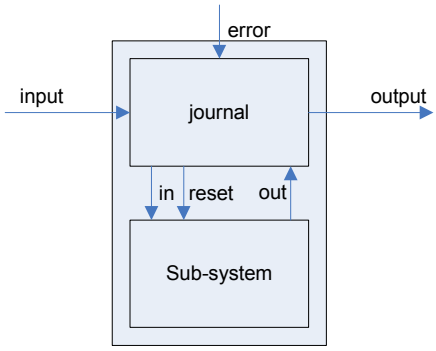
Mechanism type: Error recovery

Description: All actions are kept in a journal, enabling replay of them.

Mechanics:
The sub-system specific items are *added* to the journal. In case of problems the sub-system is *reset* and the events are *replayed*. This way the sub-system gets a "replay-able" state. Only internally important events need to be remembered, for which sub-system domain knowledge is needed. However this can also be achieved by logging all actions on the sub-system, wrapping the inputs, and saving these into the journal. This brute force approach has as special property that the sub-system doesn't have to be changed, but only be reset-able.



Domain specific journaling



Wrapper based journaling

The error detection must prevent endless journal replays. This is a second layer of defense, which is needed if the error was caused by the input sequence also kept in the journal.

Error type: Any error for which a replay brings the sub-system into a safe state.

Benefits:
- The sub-system can be brought back into the state it had before a crash.
- The wrapper variant doesn't require changes to the sub-system, besides the ability to reset.

Required system context: All side effects of replaying the inputs must be repeatable.
(E.g. launching a rocket is generally *non-repeatable* for that particular rocket)

Consequences:

Side effect: Replaying takes time, depending on the size of the journal.

Drawback: The journal can grow indefinitely for continuously running systems.

Drawback: Replaying a large journal may take (too) long.

Risk: If the input (sequence) causes the system to crash, replaying the journal will make the system crash again.

Risk: Timing aspects are lost for events kept in the journal.

Variations:

- Storing only the internal events that trigger state transitions, rather than storing all inputs, can decrease the required journal storage capacity.
- Storing only *external* inputs, making journaling sub-system independent.
- Limiting the size of the journal by only storing the inputs received *after* a known state was reached. The known state is remembered in a Checkpoint (Section 8.3).
- Timing information can be kept in the journal for each recorded event.
- Storing only the *shortest path* to each state in the journal.

Alternatives: Checkpoint and Restart (Section 7.1)

References: -

9. Looking back and forward

9.1. Conclusion

From our journey through fault tolerance literature, interviews with experts, and while consolidating our findings in this document, we have drawn the following conclusions:

- **Software fault tolerance is derived from the hardware fault tolerance domain.** Names and classifications are re-used. However, there is a difference: Fault tolerance in the hardware domain is largely focused on wear-out and electro-/magnetic interference (with solutions like spare parts), while software faults are generally programming errors (bugs). Unfortunately, in software the traditional solution — duplicating software components — also duplicates the faults, and cannot be applied straightforwardly.
- Today, **new methodologies** for requirements specification, software development, new programming **languages** (with explicit exception handling or Aspect Orientation) and rich **Operating Systems**, are increasing awareness of fault tolerance issues.
- We could not (yet) find **specific HVE mechanisms**, nor any institute[19] openly pursuing such research. Most mechanisms however, are applicable to HVE, already today, provided they are applied to select parts of the system: those that are most sensitive to failure. Integration of fault tolerance technology throughout architectures is still under research: it will challenge the seemingly unrestricted resource requirements (redundant manpower, software, hardware) of the techniques found in literature. On the plus side, Moore's law is changing the balance to our advantage.
- We expect off-the-shelve fault tolerance functionality to become **available within Operating Systems**. Linux seems the platform used to prototype such functionality. For example a snapshot of a process' memory space can be checkpointed with supplementary Linux components.

9.2. Future work

We think that the following topics in the dependability domain need more research:

- A **classification of errors**, and their relationships to faults and failures.
- Extend the inventory in this document into a **fault tolerance cookbook** for developers: step-by-step recipes to dependable systems, covering implementation guidelines to integrate off-the-shelve fault tolerance solutions.
- Determine appropriate fault tolerance levels for the HVE (or CE) domains, as introduced in Section 3.2.

[19] other than Philips Research and TRADER

- Determine the **requirements** that are representative for HVE / CE applications, with respect to fault tolerance.
- Develop a **system-wide architectural approach with regard to dependability**. For example, one must decide which parts of the system may handle faults locally, or which faults should be handled system-wide.

9.3. Open issues

This document has been written during 2005. At the moment of publication the authors still see some possibilities for its improvement:

- We would like all diagrams to follow the same notation, e.g. UML.
- It is unclear if the masking mechanisms in this document are the most important ones. We note that only a few patterns in our inventory recover by masking.
- A graph that depicts the dependencies between the patterns could guide the reader in selecting patterns. For the same purpose, a matrix that relates patterns to error types would be useful.
- A literature guide of the essential reading material on fault tolerance could be added.

Bibliography

[Laprie] *"Basic Concepts and Taxonomy of Dependable and Secure Computing"*, A. Avizienis, J. Laprie, B.Randell, C. Landwehr, IEEE Transactions on dependable and secure computing, Vol.1, No 1 January-March 2004

[FTBOOK] [Lyu 95] Michael R. Lyu, editor, *"Software Fault Tolerance"*, John Wiley & Sons, 1995

[BOOK] [Lyu 96] Michael R. Lyu, editor, *"Handbook of Software Reliability Engineering"*, IEEE Computer Society Press, McGraw-Hill, ISBN 0-07-039400-8, 1996

[NASA] *"Software Fault Tolerance: A Tutorial"*, Wilfredo Torres-Pomales, Langley Research Center, Hampton Virginia, NASA/TM-2000-210616

[Nelson 90] (uit NASA) Victor P. Nelson, *"Fault -tolerant Computing: Fundamental Concepts",* IEEE Computer, July 1990, pp 19-25

[Anderson81] (uit NASA) T. Anderson and P.A. Lee, *"Fault Tolerance: Principles and Practice",* Prentice/Hall. 1981

[Postma] *"Classes of Byzantine fault-tolerant algorithms for dependable distributed systems"*, Andre Postma, Proefschrift / Thesis, 1998, ISBN 90-365-1081-3

[HAMMING] *"Error Detecting and Error Correcting Codes"*, R.W. Hamming, The Bell System Technical Journal, April 1950

[RAND95] [Lyu95] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept", in M. Lyu editor, "Software Fault Tolerance", p1-21,, Wiley, 1995.

[Avizienis][Lyu95] Algirdas Avizienis, *"The Methodology of N-Version Programming"*, in M. Lyu editor, *"Software Fault Tolerance"*, John Wiley & Sons, 1995.

[Horn1974] J.J Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randall, *"A program structure for error detection and recovery"*, Lecture Notes in Computer Science, vol.16 pp.177-193, 1974.

[Avi1977] A Avizienis and L. Chen. *"On the implementation of N-version programming for software fault tolerance during execution"*, In Proc. IEEE COMPSAC 77, pages 149-155, November 1977.

[DAN97] F. Daniels, K. Kim, and M.A. Vouk. *"The Reliable Hybrid Pattern - A Generalized Software Fault Tolerant Design Pattern.* PLOP'97, 1997.

[DeByCo] B. Meyer, *"Object-Oriented Software Construction"* (Book/CD-ROM), second Edition, Prentice-Hall International Series in Computer Science.

[McConn04] S. McConnell, "*Code Complete – A practical handbook of software construction",* second edition, Microsoft press, 2004.

[TiPSI] T. Trew, G. Soepenberg, A. Matsinger, *Checklist for identifying issues in third-party software integration.* 2003-320/04/01 V1.0.0, Philips Research, 2003.

A Brainstorm for Fault Tolerance Mechanisms

A brainstorm session was held on fault tolerance mechanisms. This appendix explains relevant results.

The theme of the session was: "*Looking for detection and recovery mechanisms*". Robert Deckers (REPA) organized this with the TRADER team: Pierre van de Laar, Rob Golsteijn, Iulian Nitescu, Jozef Hooman, Lenart de Graaf, Paul Janson. The two main questions posed to the brainstormers were:

- How can we detect an error?
- How can we recover from an error?

In the following sections the results of each brainstorm is summarized. Concluding remarks about the results of the sessions are presented in a final section.

A.1 How can we detect an error?

Bayesians Belief Networks	Cause-and-effect graphs are used for prediction. Checking the behaviour against statistical predicted behaviour.
Check reaction on (known test) input	This is the Self-test (Section 5.6)
Value propagation tools	Errors can be detected by reasoning about the value propagation (y = x^2 + 3, means y >= 3). This can be used in Reality Check (Section 5.5) and as invariant.
Police supervision / surveillance	Can we come up with software variants for this metaphor? Surveillance is done when and where wrong behaviour is expected, but the observer is not participating (an external entity) and it knows what's right and wrong. Observer modules that check subsystem(s) for wrong behaviour might be called police modules.
Speed Camera	Are there observation techniques for certain faults, that make it easy to trap offenders, like speed cameras do for speeding cars?
User complains	Can user complains be used (maybe runtime) to improve the system? This is a big future challenge because TRADER is about runtime handling of faults. But maybe user observations that detect emotions can also be used as input for a "fault manager" although it might be hard to distinguish an angry user from a football enthusiast being "disappointed" by the home team's performance.
Using special	Can we identify techniques that can do this for software run-time?

chemicals to make hidden traces visible (Crime Scene Investigation)	Compile-time there are some techniques like code smells that could be seen as a software equivalent.
Finger-printing / water-marking	This could be an option to pinpoint fault sources and identify which part has gone wrong, if each part leaves its own fingerprint. In software already *unique ID*s are used to identify involved parties. Watermarks are already used in video and audio signals to mark origins, to be able to identify the source of illegal versions. This technique can also be used to validate media or in Digital Rights Management systems.
Compare observations with (physical) law	This can be part of a Reality Check (Section 5.5). Although causality or other time related properties might be more expensive to check (because data must be collected over time and kept up-to-date and synchronized).

A.2 How can we recover from an error?

Self repairing / healing code	Which is based on the biological ability of regeneration.
Remove / capsulate the problem (amputation)	This will most likely reduce functionality and as such is not a real recovery but might be a *graceful degradation*. How can we design a system that can still function with reduced functionality when things go wrong?
Calibrate	Calibration is used in a lot of fields, where systems are adjusted to perform as wanted. Calibration can be done manually or automatically (by a feedback control system). Systems can get "out of calibration" due to fluctuations in their environment, for example mechanical wear-out or variation due to temperature changes. Calibrate (Section 6.4) is part of the inventory, and is based on the detection mechanism Self-test (Section 5.6).
Homecoming wheel	The spare wheel is a “simpler” wheel. It forces the driver to minimize his/her mileage and speed. For software this may mean that in error cases functionality might be limited to protect the system.
Replace with a prosthetic	If we look a the replacement of a biological original part with a man made part, like false teeth, glasses, lenses, or prosthetic legs or

prosthetic	arms. The question arises what is "biological" in software and what can then be "man made", isn't software always "man made"? Furthermore, the prosthetic can be seen as a simple version of the original offering similar functionality. In which case this can be seen as a *backup spare strategy* with *graceful degradation*. In case of glasses to biological performance is even improved, which might in software mean that a newer version of a replacement component might have improved specifications or additional features. In this case there is evolution in the components that have the same interface.
Scalable data (JPEG progressive)	Can also be seen as a *graceful degradation* technique. The data is constructed is such a way that at least a less quality version is contained within the data. In the perfect case the JPEG image progressively builds up from a simple version to the full end result. This approach gives a fallback possibility in case the data gets (partly) corrupted / becomes unavailable. One advantage of scalable data can be that a less quality version (of the picture in case of JPEG) can already be presented before the whole file is transported (for example in web browsers). Another advantage can be that the most important parts of the data can be better protected than the additional parts (in case of video files).
Distract (from the error)	Is a strategy, based on human/user psychology, instead of on a solution. The system draws away the user's attention, like a magician, while performing the trick (repairing, recovery, whatever is needed). The question remains, what (trick) can be done in an actual product and is this a possible (allowed) approach. For television this leaves the question what can be done to distract the viewer and are we allowed by product management to do this? (See also "Blame someone else"). Existing CE product examples are: the message "improving system performance" while restarting or making the screen blue during channel changes to mask side effects.
Blame someone else	Our own problems can be masked by presenting/disguising them as know problems caused by external factors. This can be a "distraction" while recovering. An example can be a message stating "weak signal quality" while solving internal problems although the signal is OK.
Turn error into feature	Is not a real technical solution, but mostly a marketing statement or feature explained in the manual. The (problematic) behaviour is presented as being deliberately designed.

Punish	Tries to prevent repetition of the problem by punishing the one that caused it. It will probably be hard to design a system based on this principal, especially if it needs to keep working at the same time.
Painkillers	Dull the senses and thus hide problems for the user, in the hope that the user cans till function as expected. Are there software painkillers that "dull the senses" and make the normal behaviour unaffected by the "pain signal"?
Keep sterile	Prevent (further) contamination. This is *isolation*. In software this could be implemented with for example firewalls, and virtual machines.
Choose least worst	Suggests that there are options. If a system has options this can be a way to rank possible solutions (priority schedules). This probably is only a useful approach when the system cannot deliver the requested service, in which case it is a *graceful degradation*, or *quality of service* approach.

A.3 Brainstorm session evaluation

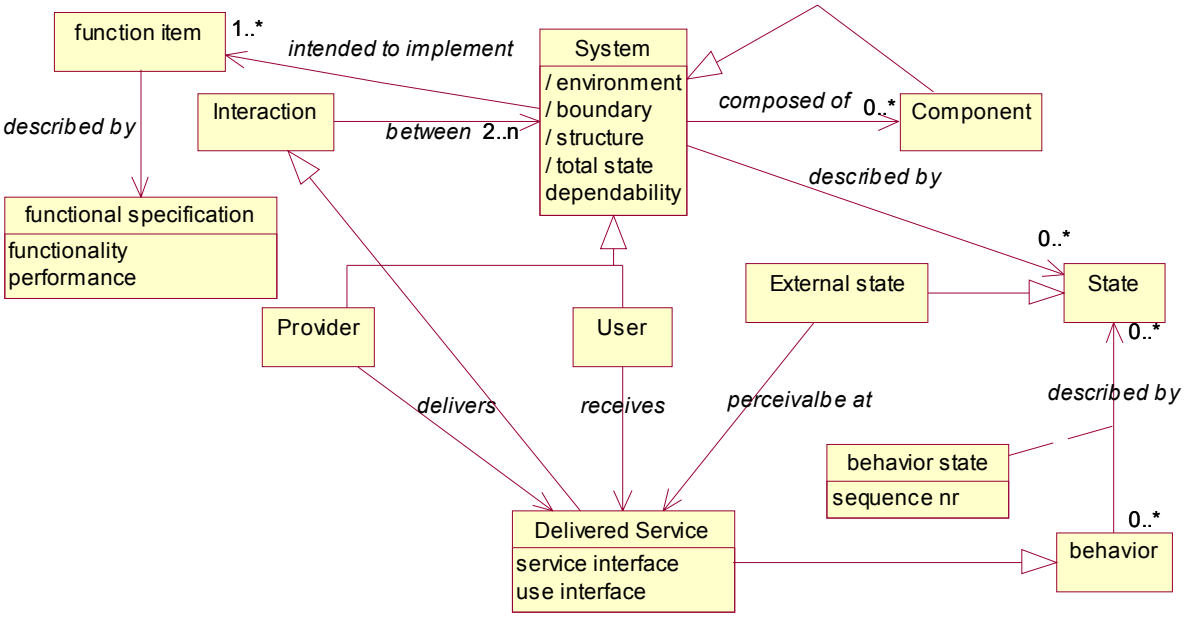
The new ideas generated in this brainstorm session can inspire new kinds of detection or recovery mechanisms. Note that some of the brainstorm items address system wide approaches, others we envision as straightforward design patterns. Even though these items are not described at the same level of detail of the patterns outlined in Chapter 4, we still consider them worthy to mention in this report.

We consider it Future Work to take up the above suggestions and develop innovative design patters or techniques.

B Concepts Model of Dependability

This appendix describes a model for the *system-* and *dependability-concepts* presented textual in the [Laprie] paper. We use the concepts of the [Laprie] paper throughout this document, and the model is meant to help to understand the paper and present an overview of the concepts. The concepts are modeled as classes in a UML class diagram and textual definitions are given below in the proposed order of reading of the model.

B.1 System Concepts

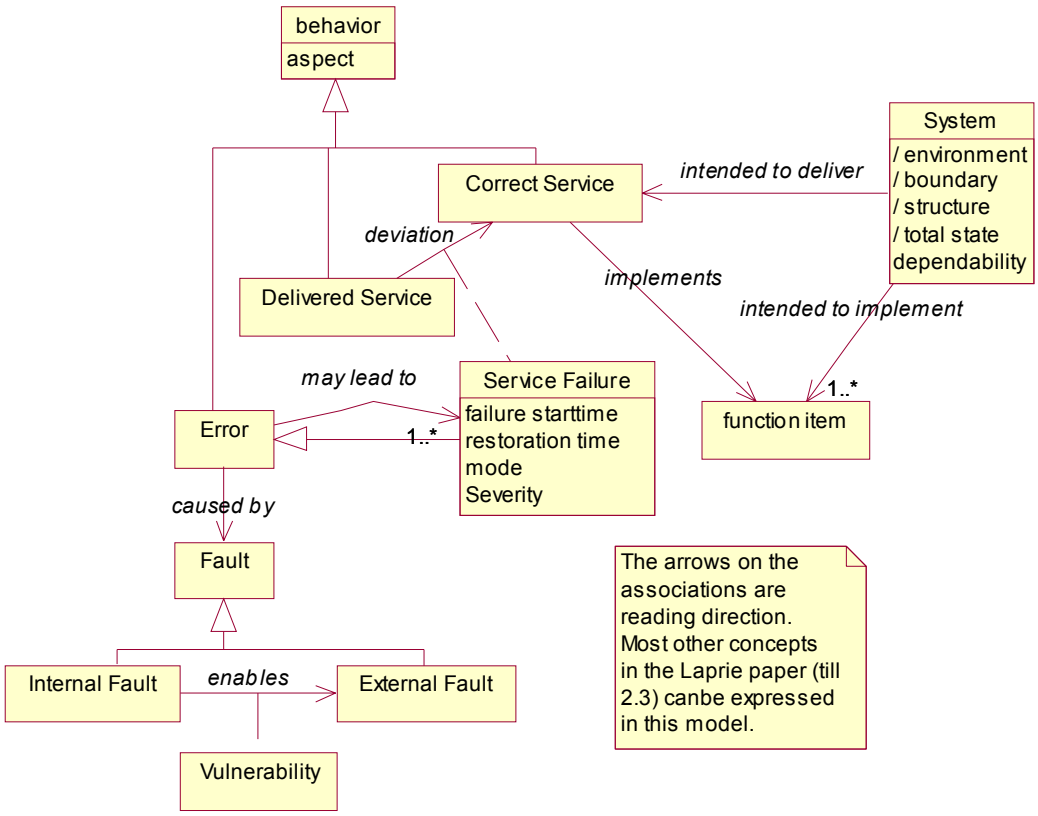


Definitions in the proposed order of reading the models picture:

- **system**: an entity that interacts with other entities, i.e., other systems.
- **component**: a system that is used in the composition of another system in order to interact with other components of that system. A system is composed of components. A component is called atomic when its decomposition is of no further interest.
- **function (item)**: that what the system is intended to do. we speak of items because the total system function can be defined by several function items.
- **functional specification**: describes a function item. (We need the difference between function item and specification because a functional specification does not always completely and correctly reflect the intended function.
- **behavior**: a sequence of states. The behavior of a system is what the system does to implement its function.
- **state**: statement about some entity, which valid for some period of time. The total state of a system is the set of external states of its atomic components.

- **behavior state**: the positions a specific state has in the sequence of states of a behavior.
- **external state**: the part of the total system a provider's total state that is perceivable at its use interface, i.e., perceivable by the user.
- **interaction**: exposures of system states to other systems.
- **delivered service**: interaction in which one **user** system perceives the function of the **provider** system.

B.2 Dependability Concepts



Definitions in the proposed order of reading the models picture:

- **correct service:** behavior that the system is intended to deliver. The correct service implements the function or a set of function item.
- **service failure**: an event that occurs when the delivered service deviates from correct service.
- **error**: part of the total system state that may lead to a failure.
- **fault**: the adjudged or hypothesized cause of an error. We speak of an **internal fault** when its origin lies within the system boundaries. Analogously, we can speak of **external faults**.

- **vulnerability**: an internal fault that that enables an external fault to harm the system.

