Recovery Design Pattern

Mohamed Fayad¹, Jayashree Rajagopalan¹, and Haitham Hamza²

Abstract—This paper introduces the Recovery design pattern. The pattern presents an abstract design model that helps in the designing the recovery problem independent of a specific application. This pattern has been developed based on the concepts of Stable Design Patterns. Stable design patterns are built based on the concepts of software stability introduced in [5,6, and 7].

Index Terms—Enduring business themes, Software stability, Software patterns, Stable analysis patterns.

I. INTRODUCTION

Recovery is a generic concept that finds applicability in numerous domains like computing, electronics, communication, medicine etc. Application of the Recovery concept varies based on the context and the domain. For e.g. the recovery of an electronic data signal is different from the recovery of password for a system. This paper aims at identifying the core knowledge behind the recovery concept that is independent of the domain specific applications. The stable pattern identifies and lists the key players in the Recovery concept, their relationships, characteristics and behaviour. The stable design pattern describes the Recovery concept from the architectural level, and hence gives the designer a clear understanding of the concept and a good starting point to design any kind of application related to recovery. Any stable pattern is unique and very specific to the concept it describes. It cannot be analogized with any other pattern or used as a template for other concepts. Any stable pattern describes an atomic concept from the architectural point of view. In case of Recovery, the pattern highlights the all the players that are linked with any Recovery process. They just include hooks, so that objects specific to any application can be tied to it. The pattern classes can be enabled or disabled based on the application. The pattern serves as a standard model for the recovery concept.

The stable design pattern for Recovery is achieved using the software stability concepts. Software stability enables a better analysis of the problem, distinguishes enduring concepts from the tangible, application specific industrial objects, captures the core knowledge, and provides a stable architecture that does not change over time.

II. SOFTWARE STABILITY AND STABLE PATTERNS: AN OVERVIEW

Software stability concept [10] is a layered approach for developing software systems. In this approach, the classes of the system are classified into three layers: the Enduring Business Themes (EBTs) layer [9], the Business Objects (BOs) layer, and the Industrial Objects (IOs) layer. Figure 1 depicts the layout of the three layers of software stability approach.

Based on its nature, each class in the system is classified into one of these three layers. EBTs are the classes that present the enduring and core knowledge of the underlying industry or business. Therefore, they are extremely stable and form the nucleus of the SSM. BOs are the classes that map the EBTs of the system into more concrete objects. BOs are semi-conceptual and externally stable, but they are internally adaptable. IOs are the classes that map the BOs of the system into physical objects. For instance, the BO “Agreement” can be mapped in real life as a physical “Contract”, which is an IO. The detailed properties that characterize EBTs, BOs, and IOs can be found in [11], [12].

Stable analysis pattern introduced in [4]-[6], is a new approach for developing patterns by utilizing software stability concepts. Stable analysis pattern was proposed as a solution for the limitations of contemporary analysis patterns in dealing with the same problems they model when these problems are encountered in different applications or domains.

The goal of stable analysis patterns was to develop models that capture the core knowledge of the problem and presented it in terms of the EBTs and the BOs of that problem. Consequently, the resultant pattern will inherent the stability features and hence can be reused to analyze the same problem whenever they appear.

¹Computer Engineering Dept., College of Engineering, San José State University
One Washington Square, San José, CA 95192-0180
m.fayad@sjsu.edu, mail_jshree@yahoo.com

²Computer Science and Engineering Dept., University of Nebraska-Lincoln
Lincoln, NE 68588, USA
hhamza@cse.unl.edu
Later on, we have generalized the concept of stable analysis patterns to accommodate design patterns as well, which result in the introduction of the new concept of Stable Design Patterns. This generalization has led to the new broad concept of stable software patterns. For the purpose of this paper, we do not need to differentiate between analysis and design; thus, we refer to both as stable patterns. The classification of stable patterns is discussed in the next section.

III. CONTEXT, PROBLEM AND FORCES

Context: Recovery is a general concept that has applicability in various domains, such as medicine, information technology etc. Recovery is a process of regaining or saving something that is lost or in the process of getting lost. For e.g. regaining health from illness, regaining data from corrupted disk etc. Even though recovery differs widely between different applications, there are some common aspects for this concept that does not change. We aim at capturing those essential features in our pattern.

Problem: Due to wide applicability of recovery pattern, it is essential to understand the core knowledge behind the recovery concept. The problem is to provide a model that captures the essential elements that any kind of recovery would embody. We try to bring out a stable pattern for recovery that can make the recovery problem much clearer by listing out the key players, their characteristics, behaviour and relationship. Such model should be an appropriate starting point for any modeler who needs to model any sort of recovery in his/her system.

Main Problem: How to build a stable recovery design pattern that can be easily extended to any kind of recovery application?

Forces:

- Recovery Pattern should resolve the following forces:
  - Recovery is a generic concept that caters to variety of applications that are totally unrelated, such as recovery from illness, recovery from a nature’s disaster. It may involve animate (such as humans) or inanimate entities (such as electronic signal).
  - The Recovery process may regain different quality in different entity. For e.g. in case of the data signal, the recovery process may regain the original data from the corrupted data, where as in human beings the recovery process may regain the health. The Recovery procedure can be performed on the same entity for different problems. Consider the case of human being recovering from his/her physical illness, also a human being recovering from a loan problem.
  - Due to the very different nature of the applications that require recovery procedure the recovery procedure will vary based on the context of the application. For instance, for a data recovery problem, we may use a noise reduction algorithm as the recovery procedure. Where as, for a heart attack illness, he/she may undergo a medical by-pass surgical procedure.
  - The Impairing Forces, any force that worsens or makes less effective the existing state of an entity, can be of very different nature like electric signal noise, malfunction of a device or external factors like germs, viruses that cause illness. The recovery pattern should be adaptable to such variety of Impairing Forces.
  - The Impairing Forces produce different kind of effect in different entities. For e.g. malfunction of a header in a hard disk drive, may corrupt the data in the hard disk, where as malfunction of a pacemaker in the heart of a human being may cause severe illness to that person.

IV. PATTERN STRUCTURE AND PARTICIPANTS

Figure 1 shows the stable model for the Recovery pattern. The model has one EBT: Recovery and Business Objects: AnyEntity, ImpairingForce, RecoveryMechanism.

```
Figure 1: Stable pattern for recovery
The participants of the Recovery pattern are:
```
Classes:
Applicability:
This is an EBT that represents the concept of being applicable. In this pattern it represents the various contexts, like Illness, Trust, Data Communication etc, where the recovery pattern is applicable. Some of its behaviour are:
- specifyProblem(): defines the recovery problem for some application in some domain.

Recovery:
This class represents the Recovery process itself. It is a pattern BO used for several and different recovery applications. It is associated with an EBT called Applicability. The behaviour of the class is as follows:
- ReceiveRequest(): The Recovery object can receive request from any entity which wishes to undergo the recovery process. Request can be sent by an AnyEntity object or the user of the system.
- FindAppropriateMechanism(): Based on the request received the Recovery object finds the appropriate instance of RecoveryMechanism to achieve the recovery process.
- InvokeRecovery(): After finding the instance RecoveryMechanism it initiates the recovery process, and executes the recovery procedures defined by that instance.

Patterns:
AnyEntity: Represents the entity that undergoes the recovery process. This object can represent entities with very different nature, such as electronic data, radio wave signal, or a file. The behaviour of this class are:
- change(): Due to some disruption it loses some of its properties and hence changes its characteristics.
- regain(): after the recovery process is completed it regains its original form.

ImpairingForce: Represents any entity that disrupts any quality of the original entity, and makes it less effective. Basically, it represents the cause of the AnyEntity deficiency. The behaviour of this class are:
- disrupt(): disrupts the original nature of any entity.

RecoveryMechanism: Represents the actual procedure followed to regain the original quality of the entity. Some of its behaviour are:
- obtainContext(): it obtains the context of the recovery problem.
- defineProcedure(): Defines appropriate procedures for recovery based on the context.

VI. APPLICABILITY

The following example illustrates the applicability of the pattern.

Case Study 1: Data Signal Recovery

Problem Statement
Modern amplifiers operate at high speeds, and often encounter unwanted disturbances in the input signal. A thermal asperity occurs when the head hits a defect or dust particle on the platter. This causes the head temperature to increase rapidly. Such a rapid and large change in temperature causes a large disturbance in the signal.

These disturbances are not part of the data signal and must be eliminated from the system for proper operation. The recovery circuit allows the amplifier to track the disturbance and adjust for it at the output. The recovery circuit has two paths. Bottom path extracts the disturbance from the incoming data, while the top path simply delays the signal the same amount of time as the bottom path. Finally, the extracted disturbance is subtracted from the delayed data. This provides an output of the original data being read, without the disturbance.
by executing the defined procedure. The DataSignal hence regains the lost quality.

**Case Study 2: Self-recovery in embedded systems**

**Problem Statement**

Consider a network of ocean buoys covering an ocean and collecting data. Each buoy can send and receive data via radio. The controllers in the buoys use Flash memory, which stores code for radio communication and data collection. The worst-case scenario for such a system is failure, crash or power outage, while the Flash is being erased or re-programmed. When uploading new code, the system should be stable enough to recover by itself, even if the system has a reset (or power failure) just right after erasing the Flash memory. To ensure the recovery, it is vital to any application that the code in the reset sector never gets erased or re-programmed.

One way to implement is using checksums. Each code segment in a re-programmable area has a predefined checksum. The code testing routine needs to calculate and match the checksum to decide if a piece of code is valid or not. For our example with buoys, the controller receives the radio command to erase the Flash memory with the data collection routines. It does execute the command, but never touches the reset sector. It now receives the new code via radio and programs it into the Flash memory. In the worst case, the system has a malfunction before the programming is complete. When the system restarts it recognizes a checksum failure in the code sector handling data collection. Using the radio link it transmits this status and waits to receive new re-programming commands.

**Class Diagram**

![Figure 4: Stability Model for Self Recovery Example](image-url)
**Dynamic Behavior**

**Sequence Diagram**

During the reprogramming process, the PowerFailure object disrupts (corrupts) the code in the programmable sector. The ProgrammableCode instance does a code validation and finds that the code is corrupted. The ProgrammableCode object sends a request to the Recovery object to undergo recovery process. The Recovery object, finds an appropriate mechanism (request for retransmission of code and reprogramming) to rectify the corrupted code. The Recovery Mechanism obtains the context of the problem and defines the appropriate procedure to rectify the problem. The Recovery object invokes the recovery process and executes the defined procedure to solve the problem. When the Recovery process is over, the ProgrammableCode object, regains the valid code and resumes its normal operation.

**Case study 3: Communications link failure recovery**

**Problem Statement**

Multiprotocol Label Switching (MPLS) Fast Reroute is an extension to MPLS that performs link failure recovery for multicast routing trees in MPLS networks. Repair of link failures in multicast communications can be performed within 50 ms (on average). The repair is done with pre-computed backup paths for an existing MPLS multicast routing tree, thus anticipating the failure. When a link is cut, traffic is rerouted quickly from the initial path to the backup path. We provide an algorithm that computes backup paths, which minimize the number of computers disconnected from the routing tree on a single link cut.

**Class Diagram**

Running the reprogramming process, the PowerFailure object disrupts (corrupts) the code in the programmable sector. The ProgrammableCode instance does a code validation and finds that the code is corrupted. The ProgrammableCode object sends a request to the Recovery object to undergo recovery process. The Recovery object, finds an appropriate mechanism (request for retransmission of code and reprogramming) to rectify the corrupted code. The Recovery Mechanism obtains the context of the problem and defines the appropriate procedure to rectify the problem. The Recovery object invokes the recovery process and executes the defined procedure to solve the problem. When the Recovery process is over, the ProgrammableCode object, regains the valid code and resumes its normal operation.
operational. The problem stalls the network and communication. The Link object sends a request to Recovery instance to rectify the problem. The Recovery object finds the appropriate mechanism (like FastReroute). The Recovery Mechanism object obtains the context of the problem. It defines procedures (like computing the back up path) to rectify the issue. The Recovery object invokes recovery process and executes the procedures to find the backup path and reroutes the data. The communication link is regained hence regains the normal network operation is resumed.

VII. DISCUSSION

As we mentioned before, the wide range spectrum of recovery applications makes the development of the stable recovery pattern a real challenge. Capturing the core aspects of the recovery concept is the fundamental player in stabilizing the resultant model. In contrast, patterns that are extracted from recovery systems that have been developed for specific applications will not scale well to accomplish the requirements of other recovery systems with different applications. For instance, a data recovery system that is developed using traditional methods most probably will be rigid enough to guarantee being collapsed when it is modified to model a recovery of a hard disk or a signal.

In this section we present the design of a simple signal recovery system. This model is then been modified to model another kind of recovery (choose anything). Though, we wish for some reusability for the common basic aspects of reuse; however, the presented model is not flexible enough to allow such reuse. Consequently, we need to design the new system from scratch.

![Figure 8: Traditional Model For Signal Recovery Example](image)

VIII. CONCLUSION

The software stability concepts form a base for building stable and reusable software patterns. The combination of the EBTs and BOs of one application can from a stable domain-specific pattern that can be reused to model similar applications. In addition, each EBT and BO in the SSM can be further modeled using stability concepts, and those models can be used to form domain-neutral patterns. The resultant patterns are stable and demonstrate efficient reusability while modeling different applications.

The examples discussed in this paper ascertain the use of the Recovery design pattern is not just analogy. The pattern can be spotted easily and it is possible to trace it back in the developed system. On the other hand, the pattern does not lose its generality, as we are able to apply it to model two different applications. Moreover, parties are built based on the share of common interest. For instance, the members of a political party do share the same beliefs upon which they are grouped into this specific party. The pattern should indicate in appropriate level of abstraction the common interest of specific party to exist.

REFERENCES

7. M.E.Fayad, V. Stanton and Hamza H. “A New Look At the CRC Cards.” [http://activeframeworks.com](http://activeframeworks.com)