An Approach to Resilient System Design using Railway-Oriented Programming

Classifications

* ACM – D.2 SOFTWARE ENGINEERING – D.2.11 Software Architectures – Patterns
* AMS – 68 Computer Science - 68N19 Other programming paradigms (object-oriented, sequential, concurrent, automatic, etc.)

Contents:

Abstract

In this paper, we formulate a set of principles and guidelines for structuring software applications leveraging the Railway-Oriented Programming pattern, in order to increase the system’s robustness

Chapter 1: Introduction

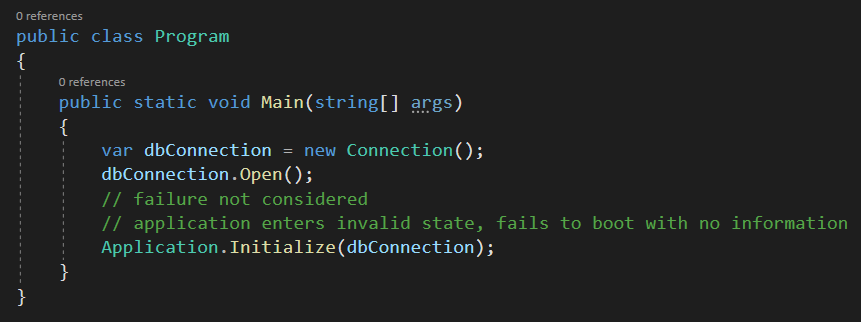
In modern times, software applications have evolved from simple, singular instances of a system designed to fulfil a single purpose or requirement, to complex webs of software systems, distributed across different hardware units, interacting over varied communication channels and managed by foreign agents. This evolution continues entropically towards the complexity singularity, increasing over time the strain on software engineers to preserve the stability of systems.

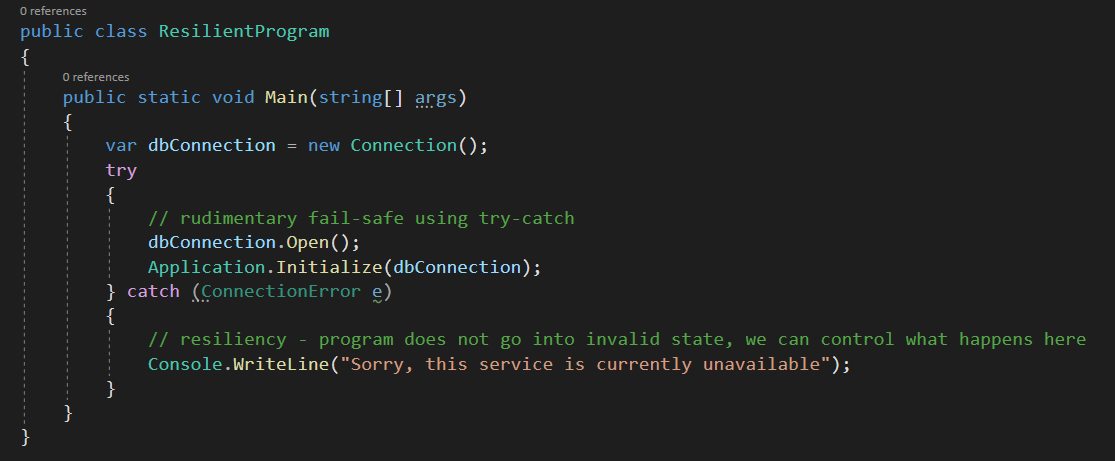
This paper aims to shape the concept of resiliency in the context of a software system, whilst proposing a concrete way of attaining it, through the use of a recently emerged programming pattern – Railway Oriented Programming (ROP). The novelty of this approach is conferred by the use of a Functional Programming Pattern – ROP, in the context of distributed system design, where Object-Oriented Programming is prevalent due to the stateful nature of inter-process communication. The resulting system uses a hybrid design, capable of leveraging the strengths of both practices.

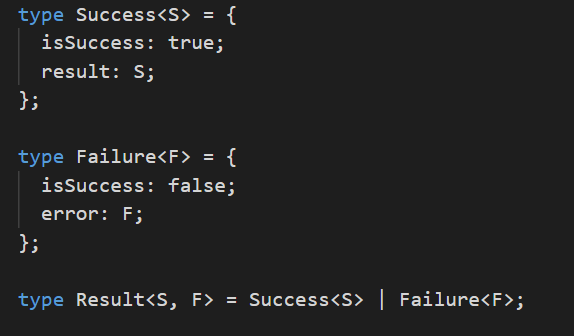
Chapter 2: Notions and Concepts

Chapter 2.1: System Design & Resiliency

System design is a doctrine within computer science focused on analyzing, managing and formulating computer software architectures.

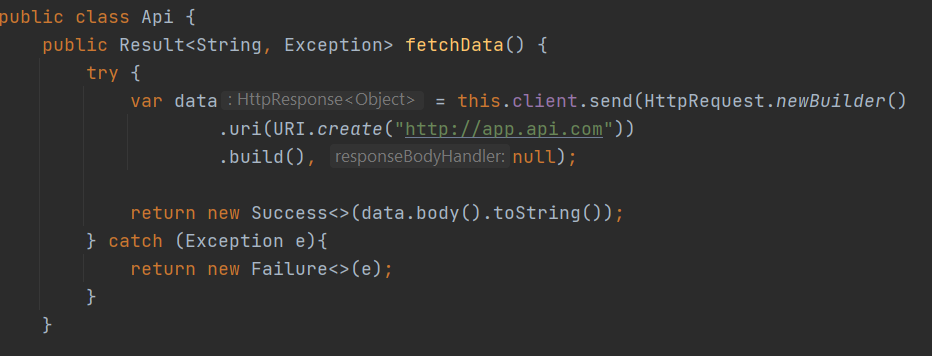
Resiliency is a characteristic of a design, used to describe how difficult it is for an application to fall into an invalid state. An application is thrown into in an invalid state as a consequence of an unexpected circumstance. Unexpected circumstances are varied, however they usually represent events which sway from the projected execution flow, or ‘happy path’, they can be caused by hardware failure, injection of invalid data, untreated edge cases and others. However, the system can only be thrown into an invalid state if these are considered ‘unexpected’, when the system does not have a projected behavior towards these events, making the outcome unpredictable, and in most severe cases, debilitating. A simple example can be seen below.

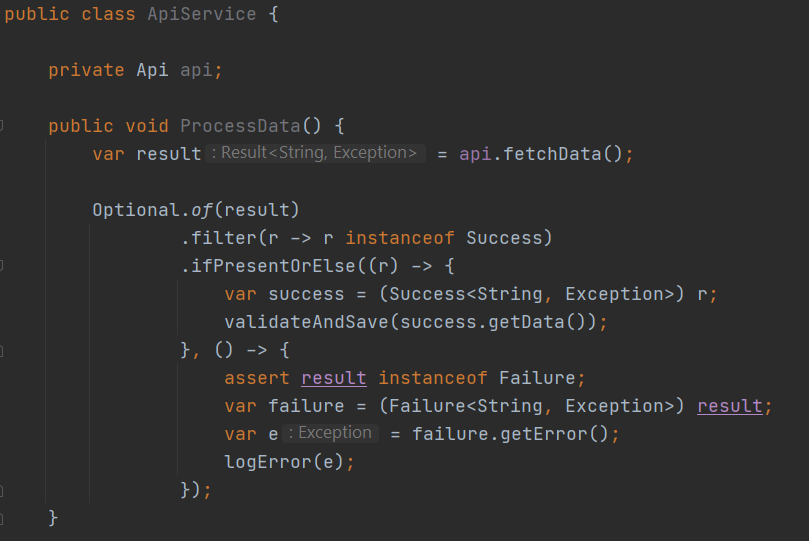


Chapter 2.2: Railway Oriented Programming (ROP)Railway Oriented Programming is a functional programming technique, originally presented in 2014 by Scott Wlaschin. It is a functional pattern that renounces the idea of having a singular main execution flow, a single ‘happy path’. This is achieved through the core concept of ROP – using a polymorphic container object that can represent two, different concepts. The mainstream implementation of ROP focuses on success and failure, in different forms, as exemplified below.

In the Typescript example above we can observe ‘Result’ being the container, while ‘Success’ and ‘Failure’ are the different concepts encapsulated within the result. The ‘Success’ and ‘Failure’ serve only to contain metadata in order to identify the nature of the result, whilst the concrete nature of the information travelling on the paths is left to be filled in, in this case using generic parameters.

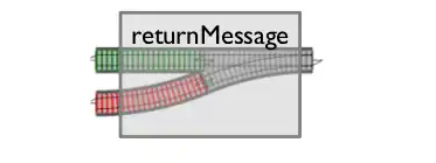
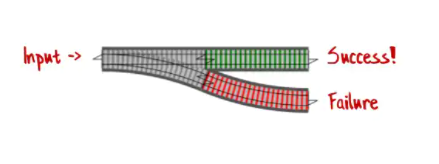
The most prevalent use case of ROP is to include error handling within the normal functional pattern, as exemplified below.

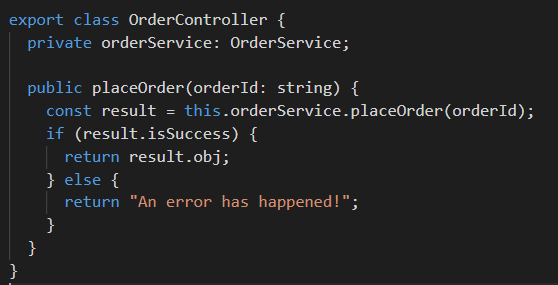
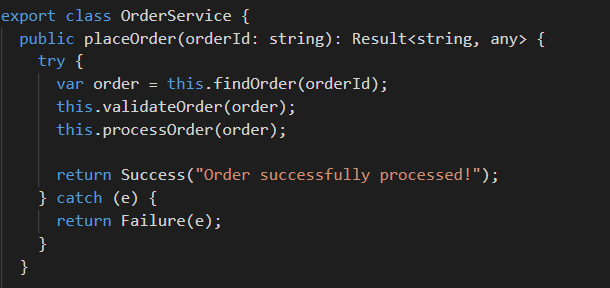




Chapter 3: The Approach

Chapter 3.1: Incremental Adoption

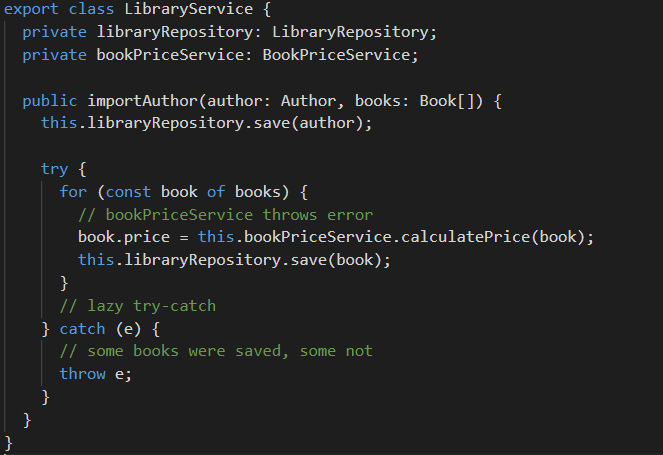
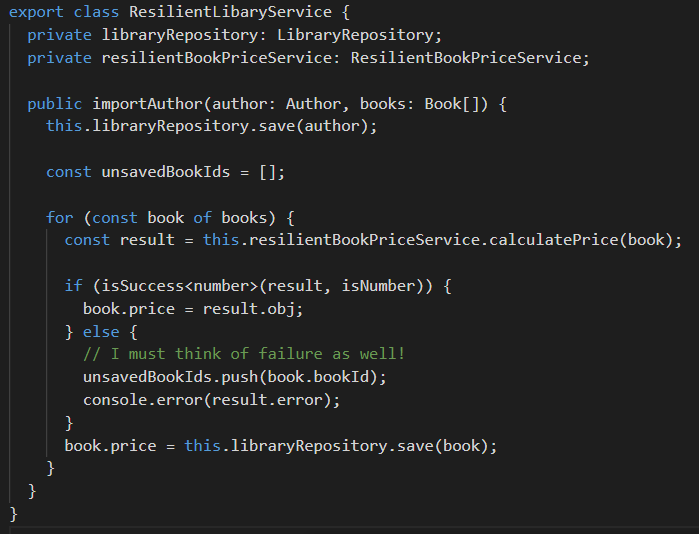
One of the biggest strengths of ROP is its flexibility. ROP has interoperability with the normal, single ‘happy path’ programs, through the use of ‘adapters’ or ‘switches’. These are the software allegory of railway track adapters, serving to convert code to, and from ROP to conventional execution flow. Conceptual example:

This means that we can incrementally introduce ROP in any part of the codebase. In the context of resiliency, this means that we can introduce ROP in a system chokepoints, where it makes sense to do so.

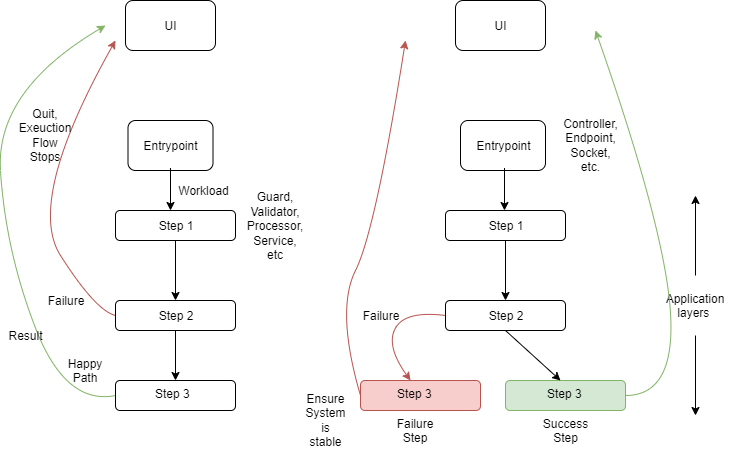
This allows us to follow a top-down approach towards the end goal of converting an entire codebase.

Chapter 3.2: Building on top of Failure

Once a codebase has integrated ROP, the possibility of failure is ingrained within the system. Global error handling, like in the above example, becomes a ‘code smell’. The error handling control flow is no longer separate from the expected execution flow, and issues generated by error handling become more apparent and must be handled.

Chapter 3.3: The Architectural View

Although the ROP technique is efficient on its own in conferring greater resiliency, its usage does not necessarily impact the system as a whole. In essence, an originally linear-flow designed project which just uses ROP is, fundamentally, still linear. In order for ROP to be used most effectively, the broader system should be reviewed , rethinking the execution flow in the context of multi-track programming.



Chapter 4: Further Considerations

The approach we have detailed in this paper is a novel way of programming execution flows, considering multiple possible execution flows, a critical one being the failure flow. The resulting model appears to be an improvement on the old one, however to assume the approach as such is nothing but shallow. The new model boasts increased complexity, which by its nature can increase both processing overhead as well as mental overhead. To add, increasing complexity is a process which must be done carefully, as inadequate application can lead to unexpected results sometimes worse than before the change.

Several caveats to this approach must be considered, as this set of guidelines is neither all-encompassing nor an absolute truth:

Multi-track programming is not meant as replacement of conventional error handling, it is designed as an added layer of abstraction on top of it, enabling the program to consider the consequences after an exception has occurred.

The ROP approach does not prove useful in cases in which the execution is not meant to continue, this includes code logic related to validation, authentication or guarding, in which the favorable approach is to fail-fast, neither is it useful in cases in which the result is ignored in the case of failure.

And finally, ROP introduces additional complexity, which is not favorable for performance-sensitive code.

Chapter 5: Testing

In order to validate the benefits of the approach, as it is a set of very specific guidelines, it must be applied in different contexts, with different requirements, and then the approach must be analyzed to conclude effectiveness. The correctness of the use of the approach must also be weighed when considering the results.

As such, we propose the introduction of the approach in the context of:

An e-commerce application

A machine learning pipeline feeder application

A microservice handling a 3rd party integration

\*\* Testing examples missing as this was previous lab \*\*

Chapter 6: Conclusion

Finally, to conclude on the results analysis, we can infer that this approach proves to be situational, with risks outweighing the benefits in some cases. We can conclude that:

* In the context of an e-commerce application, the ROP approach can improve resiliency and the overall user experience in some cases when coordinated with the UI, while in others It is used as a too complicated exception handling mechanism
* In the case of the machine learning pipeline feeder, the domain model is the center-point of the application, resiliency is key, and the approach can is highly valued
* When used to design the microservice handling a 3rd party integration, the approach is not favorable, as it adds unnecessary additional overhead, due to the nature of the execution flow being rigidly linear, whilst providing some resiliency

As such, this set of guidelines is to be regarded as a tool in a programmer’s toolbox, with a specific purpose, and must be utilized adequately, judging the benefits as well as the risks of its use.

Chapter 7. References

[1] Scott Wlaschin: Domain Modeling Made Functional, January 2018

[2] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, Yuanyuan Zhoua: Understanding the propagation of hard errors to software and implications for resilient system design, ACM SIGPLAN Notices, Volume 43, Issue 3, March 2008

[3] Stephen S. Yah, Jeffery J.-P. Tsai: A Survey of Software Design Techniques, IEEE Transactions on Software Engineering Volume SE-12 Issue 6, June 1986

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland and M. Stal: Pattern-Oriented Software Architecture, ISBN 978-0-471-95869-7 July 1996

[5] R Bird, P Wadler: An introduction to functional progamming, ISBN-978-0-13-484189-2, January 1988

[6] D.A. Turner: Total Functional Programming, Jounral of Universal Computer Science, vol 10, no. 7 (2004), 751-768

[7] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, Subhasish Mitra: ERSA: Error Resilient System Architecture for probabilistic applications, IEEE, March 2010

[8] Craig G. Rieger, Daivd I. Gertman, Miles A. McQueen: Resilient control systems: Next generation design research, IEEE June 2009

[9] John T. Taylor, Wayne T. Taylor: Software Architecture, Patterns in the Machine pp 63-82, January 2021

[10] Hans van Vliet, Antony Tang: Decision making in software architecture, Journal of Systems and Software, Volume 117, July 2016, Pages 638-644