# IMPLEMENTATION OF REDUNDANCY PATTERNS USING AOP TECHNIQUES

**Péter DOMOKOS**
**Advisor: István MAJZIK**

## I. Introduction

Software fault tolerance techniques are designed to allow a system to tolerate *software faults*. Software fault tolerance techniques are based on *design diversity*, that is, critical functions of the system (or even the entire system) are implemented by several *variants* designed and implemented by independent developers possibly using different design techniques and tools. An *adjudicator* is used to accept or to reject a result. An adjudicator can e.g. compare the results of different variants (*voter*), or implement an *acceptance test*. An *executive* orchestrates the variants and the adjudicator to realize the behavior of the fault tolerance (FT) pattern. FT patterns are e.g. *NVP* (N-Version Programming) and *RB* (Recovecry Block).

Aspect-oriented programming (AOP, [1]) is an emerging programming paradigm that tries to overcome the weakness of object-oriented programming (OOP) by modularizing features that crosscut the boundaries of objects into *aspects*. *Join points* are points of the original program code (*core concern*), where additional code (an *advice*) is executed. Advices are the implementation of the *crosscutting concerns* modularized in the aspects. An advice may be a *before* advice (executed before the join point), an *after* advice (executed after the join point), or an *around* advice. An around advice is executed instead of the code at the join point. In this case, the *proceed* pseudo-method can be used to execute the original code at the join point within the around advice. Join points are designated by special language constructs (*pointcuts*). AspectJ is an implementation of the AOP concepts for the Java language [2]. AspectJ allows to write Java programs that use AOP constructs.

In this paper, we discuss the injection of FT patterns into legacy software using AspectJ. First, a small subset of the concepts is introduced, then some aspects and the experiences of a pilot implementation are presented.

## II. Implementation of FT Patterns Using AOP

In this section, the main problems and proposed solutions are introduced that arise during the implementation of FT patterns in legacy software using AOP.

The implementation of the critical functionality is supposed to be available via a single method, called *FT method* in the following. (Otherwise, the code can become overloaded by pointcuts and difficult to overview.) An *around advice* is created that is executed instead of the FT method. This advice implements the executive (the logic of the FT pattern), that is, it orchestrates the variants and the adjudicator. If there is an acceptable result, it returns that result to the caller, otherwise, the failure is indicated. If the original code implements error handling, the failure can be indicated the same way as the FT method does it (e.g. returning null or throwing an exception of the same type).

Also the FT method must be modified, if it influences the program state or flow *outside* the FT method, e.g. by sending messages or terminating the program. This behavior is normal in the case of legacy software (e.g., in case of an error the program is terminated). However, in the case of the FT system, this is not acceptable. Therefore, error handlers must be analyzed, and possibly suppressed using an around advice that does not call the *proceed* method.

The variants must not take actions that affect the state of the program or its environment outside the variants (or, it must be ensured that the state can be restored before the execution of the next variant).

For this purpose, outgoing messages must be buffered by the executive and only sent if the adjudicator accepts the results (and in this case, the message must be sent exactly once, and not by all variants).

If the variants use volatile data (e.g. random numbers, incoming messages), it must be ensured, that all variants receive the same data. For this purpose, these data must be buffered and forwarded to the variants by the executive.

If the variants use global data, either checkpointing must be used in order to be able to restore the state after the execution of the variant, or the global environment must be emulated to the variant. This is necessary because all variants must be executed in the same initial environment, and in case of successful execution, the global environment must be modified by exactly one variant.

## III. Pilot Implementation

To examine to usability of the concepts, a pilot implementation was made. Sun Microsystems' Open Service Gateway Initiative (OSGi, [3]) implementation, called JES (Java Embedded Server) provides a framework for multiple applications (called bundles). One of the applications is the HTTP service. This service was made fault tolerant by the application of the RB pattern. The original implementation is referred to as *jesHTTP*, while the FT implementation is referred to as *ftHTTP*.

The bundle must take some actions when it is started and when it is shut down. These actions are re-implemented in the ftHTTP, and supressed in the jesHTTP. The HTTP bundle provides an interface to other bundles to register and unregister servlets and static resources that will be made available through HTTP. The implementation of this interface is provided by the ftHTTP, which forwards the requests to all variants so that all variants can keep track of the currently registered servlets and resources.

The entire HTTP service was made fault tolerant, not only parts of it. Therefore, the prerequisite that the legacy variant is available through a single method, was violated. As a consequence, several parts (related to global data, starting up and shutting down the application) of the jesHTTP had to be re-implemented in the ftHTTP.

## IV. Conclusion

According to the experiences, AOP can be used for the implementation of fault tolerance techniques in legacy software with certain restrictions. Such a restriction is that source code must be available (however, this requirement does not stem from the use of AOP).

The legacy code consist of 6.9KLoc (33 classes). The code was modified directly at 19 points in order to be able to access some classes and members, that is, only the visibility of some classes and fields had to be modified. No other direct modifications were performed on the original source code.

All other necessary modifications were carried out using aspects. 5 structural modifications were made by the introduction of fields in order to make navigation possible, and 15 behavioral modifications were made to prevent or modify the execution of an action.

The reusability of the resulting code (aspects) is restricted to the core logic of the FT pattern and the application requires extensive analysis of the original code. However, this originates mainly in the fact that the components of the redundancy pattern must be customized to the application, and is not the result of the use of AOP. The extensive analysis can not be avoided if the original code is modified directly.

## References

[1] G. Kiczales et. al., "Aspect-Oriented Programming" In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer Verlag, 1997.

[2] AspectJ, http://www.aspectj.org

[3] Open Service Gateway Initiative, http://www.osgi.org