Epi-Aspects: Aspect-Oriented Conscientious Software

Sebastian Fleissner Elisa Baniassad

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{seb, elisa}@cse.cuhk.edu.hk

Abstract

Conscientious software is a recently proposed paradigm for developing reliable, self-sustaining software systems. Conscientious software systems consist of an allopoietic part, which encapsulates application functionality, and an autopoietic part that is responsible for keeping the system alive by monitoring the application and adapting it to environmental changes. Practical application of the conscientious software paradigm requires solutions to two open problems: The design of suitable autopoietic programming languages and the proposal of concrete architectures for combining the autopoietic and allopoietic parts. In this paper, we tackle the second challenge, and propose a concrete, aspect-oriented architecture for realizing conscientious software. Here, we introduce epi-aspects, a construct for upgrading new and existing applications into conscientious software. This paper provides the architectural design of epi-aspects, an autopoietic simulator, and a concrete framework for developing epi-aspects in Java. The framework and the simulator are used to conduct a case study in which we develop and test a conscientious Java application.

Categories and Subject Descriptors D.2.11 [*Software/-Software Engineering*]: Software Architectures

General Terms Reliability

Keywords Aspect-Oriented Programming, Conscientious Software

1. Introduction

Conscientious software is a new paradigm for developing reliable, self-sustaining software systems proposed by Gabriel and Goldman in [9]. Unlike other approaches for self-sustaining software, such as IBM's autonomic computing [13, 17], conscientious software consists of two distinct parts: An allopoietic[1] part, which encapsulates application functionality, and an autopoietic[2] part, which continuously re-creates itself and is entirely devoted to keeping the system running smoothly.

The allopoietic part encapsulates traditional application functionality. It is written in a general purpose programming language, such as C++ or Java, and produces some computational results or provides services to users. The autopoietic part monitors and adapts to environmental changes, and observes and evaluates the health of the allopoietic part. In case the allopoietic part fails, the autopoietic part assists with error recovery. Error recovery and monitoring are well-understood concepts, but its techniques are not frequently applied in practice. One idea behind the conscientious software paradigm and the introduction of autopoietic / allopoietic parts is to encourage developers to devote equal efforts to implementing functionality and error recovery.

To maintain the health of the application, the autopoietic part must be able to observe and affect the operation of the allopoietic part. In [9], Gabriel and Goldman propose the concept of *epimodules*, which serve as a bridge between the autopoietic and allopoietic parts. Epimodules are attached to allopoietic components and monitor their behavior. When necessary, epimodules can affect and alter allopoietic components. For example epimodules can instruct allopoietic components to run tests, restart, upgrade, clone, or kill themselves.

The ideas and visions described in Gabriel and Goldman's paper are on a conceptual and theoretical level. In order to realize conscientious software, at least two open problems have to be solved. Firstly, new autopoietic programming languages, which are designed to prevent bugs that lead to program crashes, have to be realized to ensure that the implementation autopoietic part does not fail. Secondly, concrete,

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

[1] "Allopoiesis is the process whereby a system produces something other than the system itself. One example of this is an assembly line, where the final product (such as a car) is distinct from the machines doing the producing. This is in contrast with autopoiesis." (From www.wikipedia.org)

[2] "Autopoiesis literally means auto (self)-creation (from the Greek: auto - for self- and poiesis - for creation or production)." (From www.wikipedia.org)

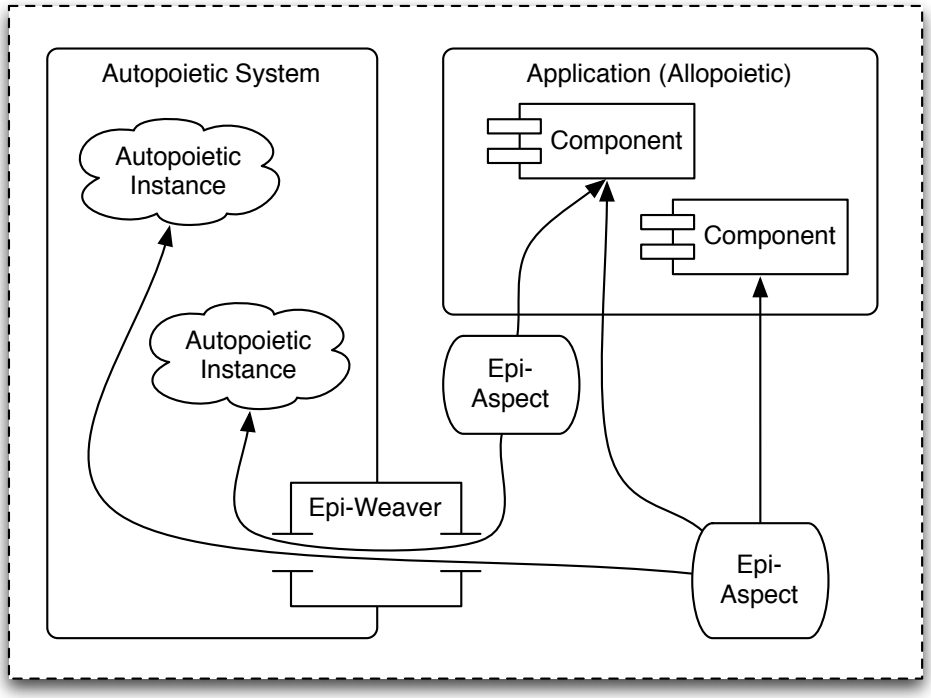


Figure 1. Aspect-Oriented Conscientious Software Architecture

practical architectures for conscientious software have to be proposed.

In this paper, we tackle the second challenge, and propose a concrete, aspect-oriented architecture for realizing conscientious software. The goal of this architecture, and that of conscientious software, is to allow the separation of the core application functionality (the allopoietic part) from the monitoring, regulation, and error recovery concerns, as provided by the autopoietic part.

Our proposed architecture (figure 1) consists of three portions: the allopoietic part (referred to as the application), the autopoietic system, and *epi-aspects*, which are the glue that binds the other two portions. Epi-aspects are able to advise on join points in both the application and autopoietic system, and so facilitate feedback from the application. They extend the application with functionality required for evaluating its health, and for performing adjustments requested by the autopoietic system. Such extensions include functionality for testing, upgrading, cloning, restarting, and killing allopoietic components. As a result, epi-aspects can be used to upgrade existing applications into conscientious software in a non-invasive manner.

This paper provides the conceptual design of epi-aspects, and proposes a concrete framework for developing epi-aspects in Java. The framework contains an autopoietic simulator that operates according to logic rules defined in the Prolog programming language. A Java epi-aspect advises on a Java application and the autopoietic simulator.

In order to evaluate the potential of epi-aspects, we perform a case study in which we use epi-aspects to upgrade an existing Java application into conscientious software. The upgraded conscientious Java application is used to conduct various tests, such as injection of programming errors, software upgrade failures, and data corruption.

1.1 Paper Organization

Section 2 describes the research context in which this paper is set. Section 3 introduces an application scenario that is used throughout the paper for illustrative purposes. This application scenario is also used in the case study in section 6 to evaluate the potential of epi-aspects. The conceptual design of epi-aspects and the architecture of aspect-oriented conscientious software are described in section 4, and a framework for developing epi-aspects in Java is introduced in section 5. Section 7 compares the epi-aspects architecture with closely related work. Section 8 discusses open problems of epi-aspects, and the paper is concluded in section 9.

2. Research Context

There are several areas of research dedicated to investigating mechanisms for allowing software systems to be self-sustaining. Among them are autopoietic software systems, autonomic computing, reflective and adaptive middleware, monitoring-oriented computing, and recovery-oriented computing.

Autopoietic software systems were first proposed in the 1970s [21]. They have been widely considered a computational model and applied in the field of artificial intelligence [22, 15]. During the 1980s the concept received less attention and was rediscovered in the late 1990s [16]. Since autopoiesis is widely considered a computational concept, most research focuses on algorithms and simulations of simple autopoietic systems.

Gabriel and Goldman describe the paradigm of conscientious software in [9], expanding on the concepts of autopoietic software systems. According to their description, conscientious software consists of an allopoietic part, which encapsulates application functionality, and an autopoietic part, which is responsible for keeping the system alive. Fleissner and Baniassad propose a concrete autopoietic-allopoietic software architecture, which is based on the biological concept of commensalistic symbiosis, in [8].

In a related effort, IBM devised the notion of autonomic computing [13, 17]. It refers to concepts and technologies that enable software to become more self-managing. To achieve this goal, autonomic computing proposes four principles, which are self-configuration, self-healing, self-optimization, and self-protection. According to [13], self-configuration refers to software components and systems that automatically follow a set of high-level configuration policies. In case of policy changes, the entire system adjusts itself automatically. Self-optimization is a process in which components continually seek opportunities to improve their own performance. The self-healing process allows the system to automatically detect and repair software and hardware problems and the self-protection mechanism defends the system against malicious attacks and failures. The self-protection mechanism uses an early warning system that allows anticipation and prevention of system failures. Researchers are exploring aspect-oriented approaches for realizing autonomic computing. For example, Engel et al. propose the usage of dynamic operating system aspects for realizing autonomic software in [7], and Greenwood et al. describe how to use dynamic aspects for implementing an autonomic system in [12].

Recent research in the field of reflective and adaptive middleware [2, 20, 14, 6] shares some goals with autonomic computing and conscientious software. As described in [11], openness and dynamic self-adaptation are fundamental properties of reflective middleware, and therefore, reflective middleware is suited to support autonomic computing and self-sustaining systems. For instance, the approach by Rasche et al. [19] proposes the usage of dynamic aspect weaving for reconfiguration. The Rainbow framework proposed by Garlan et al. in [10] is a concrete adaptive middleware architecture that uses monitoring and constraint evaluation for adaptation.

Monitoring-oriented programming, as described by Chen et al. in [3, 4, 5], is a practical programming paradigm that uses monitoring as the fundamental principle for implementing reliable software. The formal specification of an application is used as the basis for generating a set of monitors that are integrated into the software. During runtime, these monitors observe the runtime behavior of the application and trigger user-defined routines, when a specification is validated or violated.

Recovery oriented computing (ROC), explored by Patterson et al. in [18], suggests planning to incorporate or recover from a certain class of errors, rather than trying to prevent them from arising The major aim of recovery oriented computing is to minimize the mean time to repair in case a system failure occurs. In order to enable fast recovery after a failure, ROC employs the following six techniques: Recovery experiments, diagnosis, partitioning, reversible systems, defense in depth, and redundancy.

3. Application Scenario

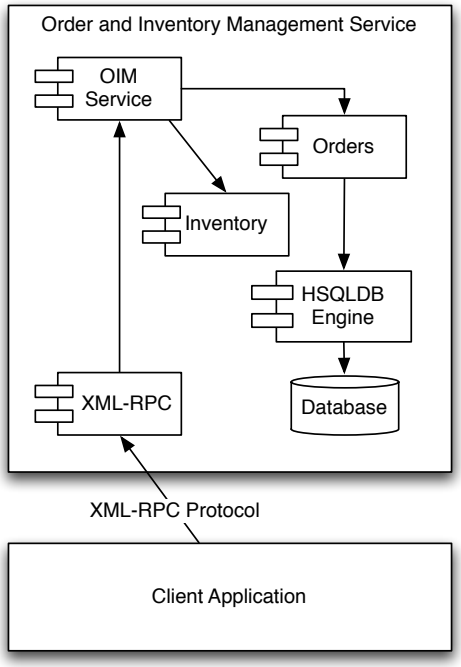


Figure 2. Application Scenario

Throughout this paper and in the case study, we consider a concrete Java application to illustrate the various features of aspect oriented conscientious software and epi-aspects. This application is a order and inventory management (OIM) system for a wholesale company that specializes in modern and antique books. As shown in figure 2, this system consists of a HSQLDB database engine, an application server, and client applications that access this server through the XML-RPC protocol. The application server contains two major components called *Orders* and *Inventory*, which implement

business rules related to managing orders and the company's inventory.

The staff of the company frequently request new features, and the system is continuously updated by a small team of developers. Also, since certain antique books do not have the same properties as modern books, such as ISBN numbers, occasional changes to the database are necessary. As a result, changes to the application server, database, and user interface of the client applications are common, and the system as a whole is constantly evolving. However, since the system is essential for the operation of the company, long down-times due to programming errors or maintenance operations are unacceptable.

The order and inventory management system is developed and improved with the focus on features that are explicitly requested by staff of the wholesale company, and mechanisms for self-maintenance and error recovery either receive a low priority or are omitted completely. As a result, the system is bound to become more fragile over time, and a complete failure is possible.

4. Proposed Architecture

The proposed aspect-oriented conscientious software architecture (illustrated in figure 1) consists of three portions:

- An application (the allopoietic system), which implements some desired functionality, such as the inventory management application described in section 3.
- An *autopoietic system*, responsible for keeping the system alive.
- *Epi-aspects*, which act as a bridge between the other two portions by extending the application with functionality related to self-sustainment, such as test routines and maintenance function.

The following sections provide a detailed description of the aspect-oriented conscientious software architecture. Section 4.1 covers the allopoietic application, section 4.2 illustrates the features and behavior of the autopoietic system, and section 4.3 describes the anatomy of epi-aspects.

4.1 Allopoietic Application

This portion of the conscientious architecture is a traditional application, such as the OIM system described in section 3. No alteration of this application is necessary since all conscientious functionality is encapsulated in the other two portions of the system. In this paper we assume object-oriented or component-based applications and we use the term *application entity* to refer to an instance of an object or component.

4.2 Autopoietic system

The autopoietic system is a network of instances that constantly observes itself, its environment, and the state of the application, which is exposed by epi-aspects. In case of any problems, such as errors in the application, the autopoietic system makes *queries*, and *recommendations* to correct the problem. The autopoietic system does not employ artificial intelligence: Software developers explicitly implement the conditions for triggering autopoietic recommendations and queries. Queries are used by the autopoietic system to monitor the health of the application. Recommendations are made either routinely, or based on the result of a query. Both recommendations and queries can be defined and customized by the developer of the autopoietic system. Our architecture provides a core set of each, as listed in table 1(recommendations) and table 2 (queries).

Start, Stop, Create, Destroy, Clone	These suggestions are directed at application entities.
Update	Recommends to update the application software. This can be directed at the whole application or application entities.
Revert	After a software update, the autopoietic system an advise the application to revert to the previous version. This recommendation is used if a problematic software update is applied.
Test	Recommends to test an application entity.
Custom	This recommendation allows the autopoietic system to advise the application to perform a custom task.

Table 1. Autopoietic Recommendations

In order to facilitate epi-aspects, which implement advice for recommendations and queries, the autopoietic system includes an internal run-time weaver for epi-aspects, called the *epi-weaver*. The epi-weaver depends on the programming language used to implement the epi-aspects, because not all programming languages provide compatible approaches for invoking methods or routines. In order to support a specific programming language, a customized weaver has to be developed and integrated with the autopoietic system.

4.3 Epi-Aspects

The purpose of epi-aspects is to make the application visible to and controllable by the autopoietic system. In particular, epi-aspects are responsible for encapsulating self-sustaining concerns and any functionality required for smooth interaction between the autopoietic system and application.

Reveal	This query indicates that the autopoietic system requires information about specific application entities or the application as a whole. For example, in order to evaluate the health of a component, autopoietic instances have to be able to examine its internal state.
Speed	This query indicates that an autopoietic system wishes to obtain information about the current speed (performance) of a certain application entity.
Custom	Allows specification of custom queries. The purpose of custom queries is to provide a flexible mechanism for extending the autopoietic system.

Table 2. Autopoietic Queries

Since epi-aspects crosscut the autopoietic system and application, they support two types of advice: The first type advises on recommendations and queries of the autopoietic system, and performs required tasks, such as testing and other maintenance. The second type advises on join points in the application and is responsible for keeping the autopoietic system updated on the application's status and performance.

Each epi-aspect contains:

1. An *epi-queue*, which is used to dispatch information from epi-aspects to the autopoietic system.
2. *Application advice*, which are responsible for providing feedback on the application's health to the autopoietic system. This feedback is passed to the autopoietic system through the epi-queue.
3. A mechanism for defining advice on autopoietic recommendations and queries. Such *query advice* and *recommendation advice* perform maintenance or update operations according to suggestions by the autopoietic system, or to provide information to satisfy an autopoietic query.

As epi-aspects implement advice for both application and autopoietic system, they have to be woven twice by different weavers: One weaver is responsible for weaving application advice during compilation or at run time. The other weaver is the epi-weaver of the autopoietic system. For example, let's consider an epi-aspect denoted as *DatabaseEpiAspect*, which is part of an upgrade to turn the OIM service introduced in section 3 into conscientious software. The application advice of this epi-aspect are woven into the code of the database engine during compile-time, and the recommendation and query advice are woven into the autopoietic system during run-time.

Sender, Receiver	Both autopoietic instances and application entity can acts as sender and receiver.
Type	Indicates the type of the epi-message. Can be one of the following: • Parameter: Epi-message is passed to an autopoietic recommendation or query advice as parameter. • Answer: Epi-message is an answer to an autopoietic query. • Feedback: Epi-message contains feedback regarding the allopoietic application. • Error: Epi-message reports an error that occurred in the allopoietic application. • Epi-Error: Epi-message reports an error that occurred in an epi-aspect.
Contents	Contains the contents of the message. The format of this message could be anything from strongly typed data over XML to natural language. The only requirement is that both autopoietic instances and epi-aspects are able to interpret this format.

Table 3. Epi-Message Attributes

4.3.1 Epi-Messages and Epi-Queue

The autopoietic system and epi-aspects communicate by exchanging *epi-messages*, which consist of the attributes shown in table 3. The autopoietic system can pass epi-messages as parameters to recommendation and query advice implemented by epi-aspects.

Each epi-aspect has access to an *epi-queue*, which can be used to dispatch information to the autopoietic system. Before an epi-aspect is woven into the autopoietic system, it merely contains a stub for the epi-queue that does not contain any functionality. When the epi-weaver processes an epi-aspect, it does not only weave recommendation and query advice, but also injects a concrete implementation of the epi-queue. This approach ensures that epi-aspects do not depend on the concrete realization of the autopoietic system.

4.3.2 Application advice

Application advice are defined on joint points in the allopoietic application. Their purpose is to observe one or more specific application entities and to expose their state to the autopoietic environment. Depending on the join point it advises on, the implementation of an application advice gathers and optionally forwards information to the autopoietic sys-

Error	Advice for errors or exceptions gather as much information on the error as possible, including the source of and reason for the error, and then dispatch this information to the autopoietic system via the information queue. For example, in the OIM service, error application advice can be defined for exceptions thrown by the database engine and XML-RPC service.
Creation	Advice for creation join points are responsible for informing the autopoietic system which application entities, such as objects, components, and modules, exist. The autopoietic system uses this information to decide which entities should be monitored.
Performance	Performance advice are invoked before and after certain methods or procedures in the application. Their purpose is to measure the execution time of methods and report it to the autopoietic system. This allows the autopoietic system to keep track of the application's performance and detect possible timeouts.

Table 4. Application Advice

tem. Information is dispatched to the autopoietic system via the epi-queue.

Existing aspect-oriented programming languages, such as AspectJ, provide sufficient pointcut primitives to describe most of the application join points required by epi-aspects. As a result, epi-aspects can be realized as an extension of existing aspect-oriented programming languages.

The following application-level joinpoints are required by the epi-aspect architecture:

- **Create**: Creation of a new application entity, such as the construction of an object.
- **Destroy**: Destruction of an application entity, such as a component or object.
- **Error**: Unexpected errors or exceptions.
- **Invocation/Execution**: Invocation and execution of methods.
- **Event**: Events, in an event handling system.
- **Set**: Setting of a variable or property value.

Table 4 provides the details of how a selection of the application advice might be used.

4.3.3 Query and Recommendation Advice

Query and recommendation advice implement maintenance and information retrieval operations proposed by the autopoietic system. They are woven into the autopoietic system by the epi-weaver at runtime.

Developers of the OIM system might implement recommendation advice to implement a unit test to verify the proper operation of the XML-RPC component, or to apply updates to the OIM system. They may implement a query advice to implement an evaluator that obtains the current speed of the XML-RPC requests per minute. That implementation would advise on the *Speed* suggestion, and would help the autopoietic system to keep track of the size of the workload on the XML-RPC component.

Query and recommendation advice consist of two portions:

- **Header**, which contains the attributes *name* and *receiver-pattern*. The *name* attribute is the name of a predefined (see tables 1 and 2) or custom recommendation or query. The *receiver-pattern* attribute contains a regular expression for matching the target application entity.
- **Implementation**, which contains the allopoietic code for implementing the requested action. It uses epi-messages as input and output parameters as a means for communication between the autopoietic system and epi-aspect.

5. Epi-Aspects Java Framework

This section specifies a framework for developing aspect-oriented conscientious software in Java. This framework, which is called *Epi-AJ*, provides an autopoietic simulator and constructs for implementing epi-aspects in the Java programming language. The autopoietic simulator includes a weaver for epi-aspects, and contains a logic engine implemented in the Prolog programming language, which implements the behavior of the autopoietic system described in section 4.2.

The Epi-AJ framework is designed as a supplement to Aspect-J. Since version 5, AspectJ has supported the usage of Java annotations for defining aspects and advice. Epi-AJ provides a set of Java annotations, which allow the definition of autopoietic recommendation and query advice. As a result, an epi-aspect can be implemented using a combination of AspectJ annotations and Epi-AJ annotations. The usage of annotations is convenient in a sense that it is not necessary to use tools like the AspectBench compiler [1] to extend the grammar of the AspectJ pointcut language with new pointcut primitives for epi-aspects. Listing 1 illustrates the definition of an epi-aspect using the combination of Aspect-J and Epi-AJ.

The Epi-AJ framework is divided into three Java packages:

1. The package *conscientious.epiaj*(shown in figure 3) contains the base classes and interfaces of the framework.

```
1  @Aspect public class XMLRPCEpiAspect extends EpiAspect
2  {
3      @After("this(s) && execution(XMLRPCService.new(..))")
4      public void newInstance(XMLRPCService s) {/*...*/}
5
6      @AfterThrowing("target(s) && execution(* XMLRPCService.run(..))")
7      public void reportException(XMLRPCService s) {/*...*/}
8
9      @RecommendationAdvice(recommendation="start", recipientPattern=".*")
10     public EpiMessage startXMLRPCServer(EpiMessage message){/*...*/}
11
12     @CloneRA("org.apache.server.*")
13     public EpiMessage cloneXMLRPCServer(EpiMessage message){/*...*/}
14
15     @QueryAdvice(query="speed", recipientPattern="org.apache.server.*")
16     public EpiMessage getCurrentXMLServerSpeed(EpiMessage message){/*...*/}
17
18     @RevealQA(".*")
19     public EpiMessage revealObjects(EpiMessage message){/*...*/}
20
21     /*...*/
22 }
```

Listing 1. Epi-Aspect Example Code (Epi-AJ Framework)

2. The package *conscientious.epiaj.annotations* contains annotations for declaring autopoietic recommendation and query advice in epi-aspects.
3. The package *conscientious.simulator* contains the Java part of the autopoietic simulator, such as the implementations of the epi-weaver and epi-queue.

5.1 Base Classes and Interfaces

As illustrated in figure 3, The Epi-AJ framework provides the following set of base classes and interfaces for realizing epi-aspects, epi-messages, and epi-queues: `EpiAspect`, `EpiQueue`, and `EpiMessage`.

As described in section 4.3.1, epi-aspects and the autopoietic system communicate by exchanging epi-messages. Each epi-aspect has access to an epi-queue that allows them to dispatch epi-messages to the autopoietic system.

The abstract class `EpiAspect` is the base class for epi-aspect implementations, `EpiQueue` defines the interface of epi-queue implementations, and the class `Epi-Message` is the implementation of the epi-message illustrated in table 3.

The `EpiAspect` class contains an instance variable whose type is the `EpiQueue` interface. When an the implementation of an epi-aspect is woven, the autopoietic system (or autopoietic simulator) assigns a concrete epi-queue implementation to this instance variable. After that, the epi-aspect implementation can start dispatching epi-messages to the autopoietic system.

The `Epi-Message` class contains four instance variables, which are equivalent to the epi-message attributes `Sender`, `Receiver`, `Type`, and `Contents` described in table 3.

5.2 Advice and Annotations

The Epi-AJ framework provides the following set of Java annotations for declaring recommendation and query advice:

- @RecommendationAdvice(name, receiverPattern)
- @QueryAdvice(name, receiverPattern)
- @RevealQA(receiverPattern)
- @SpeedQA(receiverPattern)
- @TestRA(receiverPattern)

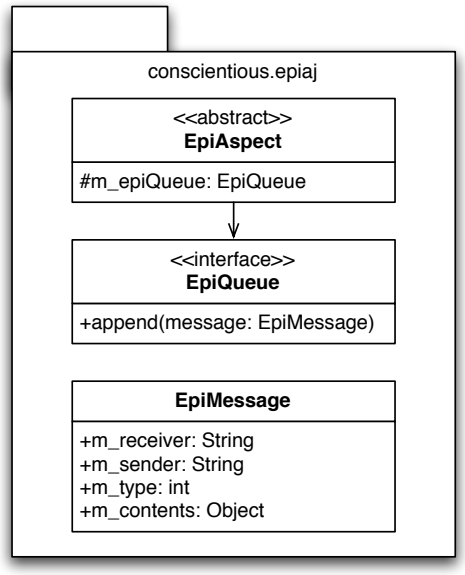


Figure 3. Epi-AJ Base Classes

- @UpdateRA
- @RevertRA(receiverPattern)
- @CloneRA(receiverPattern)
- @CreateRA / @DestroyRA(receiverPattern)
- @StartRA / @StopRA(receiverPattern)

All annotations have a `receiverPattern` attribute that can be used to specify a regular expression for matching the class/epi-aspect at which the autopoietic recommendation or query is directed. The @RecommendationAdvice and @QueryAdvice annotations are generic annotations that can be used to declare advice on any autopoietic recommendation and query, including custom recommendations and queries. The remaining annotations are provided for convenience and can be used to specify advice on the pre-defined autopoietic recommendations and queries described in tables 1 and 2.

As shown in listing 1, the implementation part of the recommendation and query advice is a Java method that receives an `EpiMessage` object as parameter and returns another `EpiMessage` object to the autopoietic system. The `EpiMessage` parameter is set up by the autopoietic system to specify details regarding the recommendation or query, and the `EpiMessage` return value contains feedback or other information for the autopoietic system.

5.3 Autopoietic Simulator

The Epi-AJ framework provides an autopoietic simulator that can be used for developing and testing epi-aspects. This simulator consists of a runtime, an epi-weaver written in Java, and uses the Prolog programming language to implement the rules of the autopoietic system. Prolog is not an autopoietic programming language that is specifically designed to prevent bugs that can lead to program crashes. However, as the design of an autopoietic programming language is not within the scope of this paper, Prolog is a suitable substitute for simulation purposes, because it is declarative and it is not easy to write a Prolog program that crashes.

The autopoietic simulator can be invoked from a Java program by creating and configuring an instance of the `Simulator` class shown in figure 4. Internally, the `Simulator` class uses the SWI Prolog engine to simulate the behavior of the autopoietic system. Interaction between the `Simulator` instance and the SWI Prolog engine is accomplished through the JPL (Java Interface to Prolog) API, which is part of the SWI Prolog distribution.

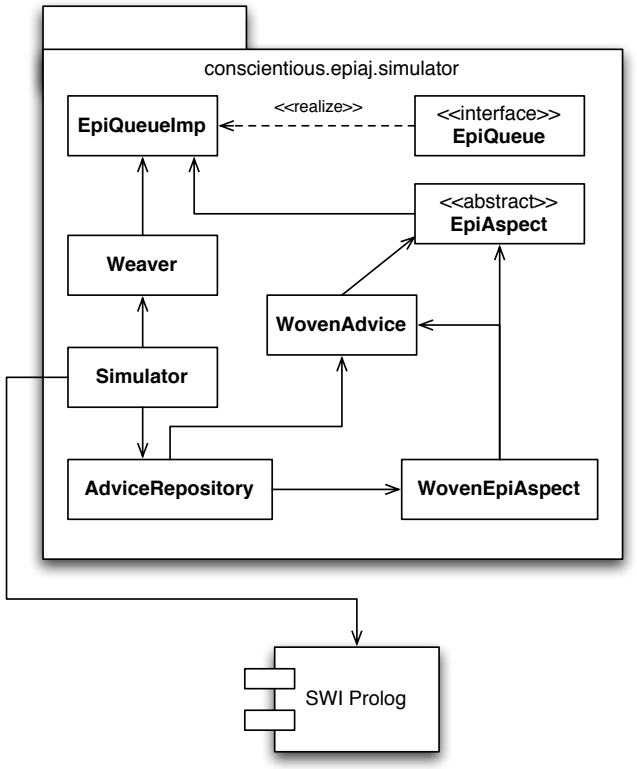


Figure 4. Epi-AJ Autopoietic Simulator

When a new instance of the `Simulator` class is created, the calling application provides a list of epi-aspects. During its initialization, the `Simulator` instance performs the following tasks:

1. The SWI Prolog engine is initialized and the Prolog program(s) mimicking the autopoietic systems are loaded.
2. An instance of the `AdviceRepository` shown in figure 4 is created.
3. An instance of the `Weaver` class is created and the list of epi-aspects is passed to it.

4. The `Weaver` instance weaves the epi-aspects into the `AdviceRepository` instance. Moreover, it injects an instance of the `EpiQueueImp` class, which implements the simulators epi-queue into each woven epi-aspect.
5. The `Simulator` instance issues the autopoietic recommendation *Start*, which is dispatched to all woven epi-aspects.

Once the autopoietic simulator is running, the woven epi-aspects can dispatch epi-messages to it via the `EpiQueueImp` instance. Whenever an epi-message is received, the Prolog program(s) are invoked and the result can be an autopoietic recommendations or query. Autopoietic recommendations and queries are dispatched to relevant advice of the woven epi-aspects.

6. Case Study

This section describes a case study that was conducted to evaluate the potential of aspect-oriented conscientious software. The first part of this study, which is described in section 6.1, illustrates how to apply epi-aspects and the Epi-AJ framework to turn the order and inventory management (OIM) system introduced in section 3 into conscientious software. The second part in section 6.2 describes experiments and studies that were conducted to test and evaluate the conscientious version of the OIM system.

Figure 5 shows a more detailed view of the OIM system's design, and highlights the main Java classes of the OIM system: `XmlRpcService`, `OrderManager`, `OimService` , `InventoryManager`, `DbConnectionPool`, `UserAccounts` and `DbConnection`.

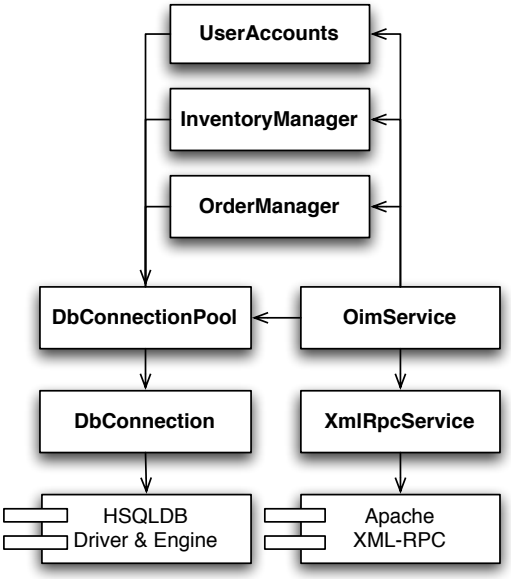


Figure 5. OIM System Classes

The `XmlRpcService` class uses Apache's XML-RPC distribution to initialize a HTTP server that accepts XML-RPC requests. This server uses reflection to map incoming XML-RPC requests to an instance of the `OimService` class. Additionally, it converts return values provided by methods of the `OimService` instance into XML-RPC responses.

Even though Apache's XML-RPC distribution is mature and stable, these classes can generate critical exceptions in case of invalid requests and network problems.

The business logic of the OIM system is implemented by the `OrderManager`, `InventoryManager`, and `UserAccounts` classes.

The OIM system uses the HSQLDB database engine. The database is accessed via the classes `DbConnectionPool` and `DbConnection`. The class `DbConnectionPool` maintains a pool of re-usable `DbConnection` instances, which provide access to the database via the JDBC driver supplied with the HSQLDB distribution.

The class `UserAccounts` implements user management and authentication.

6.1 Implementation Overview

The purpose of the first part of this case study is to use epi-aspects and the Epi-AJ framework to upgrade the OIM system into conscientious software. The aim of this upgrade is to make the OIM system observable and controllable by an autopoietic system.

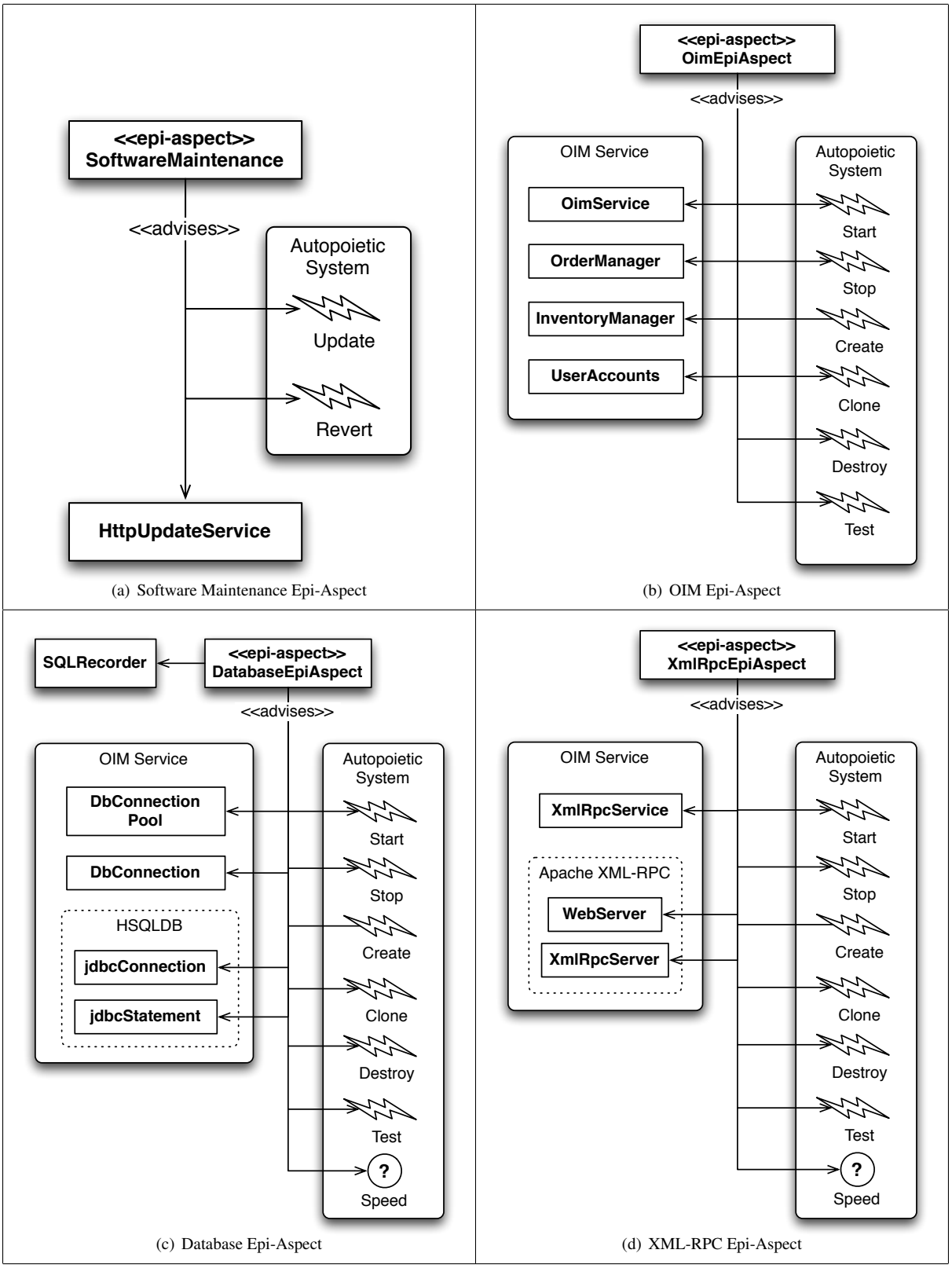
The following sections describe the implementation of four epi-aspects, which add necessary conscientious extensions to the OIM system: software maintenance, XML-RPC monitoring, Database monitoring, and OIM system monitoring. This upgrade is non-invasive, since it is unnecessary to modify the existing source of the OIM system, the HSQLDB engine, and Apache's XML-RPC distribution.

6.1.1 Software Maintenance Epi-Aspect

The software maintenance epi-aspect (figure 6(a)) implements functionality for updating and reverting the components of the OIM system. It provides advice for the autopoietic *Update* and *Revert* recommendations. When the software maintenance epi-aspect is initialized, it creates a minimal HTTP service, which developers can use to submit software updates via a web-browser.

Whenever the software maintenance epi-aspect receives an update through the HTTP service, it does not immediately install the update, but stores it for later use, and dispatches an epi-message to notify the autopoietic system that an update is available. If the autopoietic system approves of the update, it first issues recommendations to affected components to prepare for an imminent update, and then issue the *Update* recommendation, which causes the software maintenance epi-aspect to install the update.

If the autopoietic system notices that certain components experience problems after an update, such as uneven perfor-



(a) Software Maintenance Epi-Aspect

(b) OIM Epi-Aspect

(c) Database Epi-Aspect

(d) XML-RPC Epi-Aspect

Figure 6. Case Study Epi-Aspect Design

mance, it can issue a *Revert* recommendation that indicates that the problematic component should be reverted to a previous version. The software maintenance epi-aspect implements an advice on the *Revert* recommendation that checks if a previous version of the affected component exists. If a previous version is available, the advice disables the current version, and re-installs the previous version.

6.1.2 XML-RPC Epi-Aspect

The OIM system is accessed by clients via the XML-RPC protocol. The service providing this access is implemented by the `XmlRpcService` class, which utilizes Apache's XML-RPC distribution. It is imperative for the OIM system that the XML-RPC service does not fail. The XML-RPC epi-aspect (figure 6(d)) is responsible for implementing error recovery, testing and application monitoring concerns. It also implements an observer feature to evaluate and store the current speed of the XML-RPC service. The speed is defined as the time required to execute a dummy XML-RPC request.

6.1.3 Database Epi-Aspect

The database epi-aspect (figure 6(c)) encapsulates functionality that allows the autopoietic system to observe and interfere with the operation of the HSQLDB database engine. In addition, the database epi-aspect implements database backup and recovery features. The backup feature periodically backs up the database files and allows restoring the database to a previous version. It is useful for preventing problems related to data corruption. The database epi-aspect records all SQL commands that are issued to the HSQLDB engine from within the OIM system. This recorded history can be used for undoing changes to the database and its schema.

6.1.4 OIM Epi-Aspect

The OIM epi-aspect (figure 6(b)) is responsible for exposing the health of the main classes of the OIM system, namely `UserAccounts`, `OIMService`, `OrderManager`, and `InventoryManager` to the autopoietic system. Additionally, the OIM epi-aspect extends these main classes with functionality to comply recommendations issued by the autopoietic system.

6.2 Experiments and Studies

One experiment and one study were conducted to evaluate the potential of the conscientious OIM system. The experiment evaluates ability of the conscientious OIM to recover from buggy updates, and the study illustrates the process of extending the conscientious OIM system with a finer-grained monitoring and recovery mechanism.

6.2.1 Software Update Experiment

In the software update experiment, working and buggy updates are applied to the original and conscientious OIM systems, and the behavior of the systems during and after the update is observed, compared, and evaluated.

The experiment consists of two phases and each phase consists two parts:

In the first part, the experiment is conducted with the original OIM system, which does not make use of any epi-aspects. In the second part, the experiment is repeated using the conscientious version of the OIM system and the autopoietic simulator. Then the results of both parts are compared and evaluated.

The experiment uses the original OIM system, the conscientious OIM system, the autopoietic simulator, and an additional simulator that mimics the behavior of a client application that accesses the OIM system. This client application simulator can be configured to generate a specific number of requests per minute, allowing adjustment of the workload that the OIM system has to handle. The client application simulator uses a log file to record requests results and exceptions that occur when accessing the OIM system.

Phase 1: Install Working Update

A working update of the `XmlRpcService` class is installed, and the necessary steps and time required for updating the original and conscientious OIM systems are compared. This phase is initialized by performing the following steps:

1. Start the original OIM system.
2. Start the conscientious OIM system and autopoietic simulator.
3. Start two instances of the client simulator and configure them to issue one request per second to the original and conscientious OIM systems. These instances are denoted as *client simulator 1*, which issues requests to the original OIM server, and *client simulator 2*, which issues requests to the conscientious OIM server.

After phase one has been initialized, the following steps are executed in sequence:

1. Manually overwrite the Java class file of the `XmlRpcService` class in the original OIM system with the new version.
2. Shutdown and restart the original OIM system.
3. Use the HTTP update service of the software maintenance epi-aspect to submit the source code of the updated `XmlRpcService` class to the conscientious version of the OIM system.

The results of the first phase are shown in table 5. While the update is successful for both the original and conscientious OIM, the original OIM system has to be restarted which makes the system unavailable for approximately 17 seconds. This short period causes a number of XML-RPC requests issued by the client simulator 1 to fail. The con-

scientious OIM system, however, experiences no downtime, because the software update epi-aspect applies and initializes the updated version of the `XmlRpcService` class in the background and then immediate replaces the old instance with the new one. This result indicates that the conscientious OIM version is more suitable for being updated during production use.

Original OIM	Update successful. Approx 17 seconds downtime, because the entire OIM system is restarted.
Client Simulator 1	Log file indicates 19 failed requests.
Conscientious OIM	Update successful. No downtime.
Client Simulator 2	Log file indicates no failed requests.

Table 5. Software Update Experiment Phase 1 Results

Phase 2: Install Buggy Update

In the second phase a buggy update of the `XmlRpcService` class is installed. The bug in this update is a latent bug that causes critical failure after the `XmlRpcService` instance has been running for approximately one hour. This phase uses the same steps as the first phase, except that the updated version of the `XmlRpcService` class contains a latent critical bug that starts causing failures after approximately one hour. After the updates are applied to both OIM systems, their behavior is observed for three hours.

Table 6 shows the observation log of the second phase. As the observation log indicates, the XML-RPC service of the original OIM system fails after approximately 61 minutes. The conscientious OIM system restarts the XML-RPC service after it causes an initial exception in the 61st minute of the experiment. The restart prevents further exceptions for approximately one hour. When exceptions start occurring again in the 125th minute of the experiment, the conscientious OIM system reverts the XML-RPC service to its previous version and continues running smoothly until the end of the experiment.

6.2.2 Fine-Grained Monitoring Study

The purpose of this study is to illustrate the software design and development process of aspect-oriented conscientious software by extending the conscientious OIM system with a finer-grained error monitoring and recovery mechanism. This extension involves extending the OIM epi-aspect as well as writing new rules for the autopoietic simulator. Here, we describe the design and implementation, and the

Experiment Time (Min)	Event
61:23	First exception in the *XmlRpcService* instance the original OIM system.
61:27	First exception in the *XmlRpcService* instance the conscientious OIM system.
61:27	The autopoietic simulator recommends to restart the *XmlRpcService* instance of the conscientious OIM system, which is done by the XML-RPC service epi-aspect.
61:29	The *XmlRpcService* instance of the original OIM system terminates. Original OIM system not accessible.
125:14	Exception in the *XmlRpcService* instance the conscientious OIM system. Autopoietic simulator recommends reverting the *XmlRpcService* to a previous version, which is done by the software maintenance epi-aspect.
180:00	Conscientious OIM system is still running properly.

Table 6. Software Update Experiment Phase 2 Log

approach for testing the extended conscientious OIM system.

The finer-grained error monitoring and recovery mechanism maintains a history of exceptions for the each of the classes `UserAccounts`, `OIMService`, `OrderManager`, and `InventoryManager`. The functionality for the exception history is implemented by a component that is added to the OIM epi-aspect. Whenever an exception occurs in one of the OIM main classes, this component makes an entry into a log file associated with the class that threw the exception. Apart from creating the entries, the component counts the number of exceptions in pre-defined intervals. The number of exceptions per interval is recorded and submitted the numbers of exceptions for the five most recent intervals are submitted to the autopoietic system via the epi-queue.

The autopoietic simulator requires additional rules for processing the epi-messages containing the exception counts of the five previous intervals:

1. If the number of exceptions for the current interval is greater than zero, issue a *Test* suggestion. The corre-

sponding advice in the OIM epi-aspect verifies the responsiveness of the OIM system classes. If the test fails, the default rules of the autopoietic simulator trigger a *Restart* suggestion.

2. If the number of exceptions between the four previous and current interval have increased more than a predefined threshold, issue a *Restart* suggestion.

To test the proper operation of the implementation of the fine-grained monitoring feature, updated versions of the classes `OIMService` and `OrderManager`, which randomly throw non-critical and critical exceptions, are added to the conscientious OIM system. Non-critical exceptions do not affect the proper operation of instances of these two classes, and critical exceptions lead to a crash of the OIM system. The test is run according to the following protocol:

1. The updated OIM epi-aspect, autopoietic rules, new versions of `OIMService` are `OrderManager` are deployed in the conscientious OIM system.
2. The component in the OIM epi-aspect is configured to use intervals of three minutes.
3. The autopoietic simulator is configured to use 10 exceptions as the threshold value for triggering *Restart* suggestion.
4. One instance of the client simulator is started with the same configuration used in the software update experiment described in section 6.2.1.
5. Let the experiment run for 120 minutes.

This test can only fail if there is an implementation error in the OIM epi-aspect. After the test is run successfully, the development of the fine-grained monitoring feature is completed.

7. Comparison With Related Work

The goals of self-sustainment and self-adaptation are not limited to conscientious software. As described in section 2, autonomic computing, monitoring oriented programming, and reflective and adaptive middleware attempt to tackle the same issue using various approaches. This section compares epi-aspects with concrete architectures proposed within these fields.

7.1 Reflective and Adaptive Middleware

The Rainbow [10] and Adapta [20] frameworks are concrete adaptive and reflective middleware architectures.

The Rainbow framework by Garlan et al. consists of an architecture layer, a system layer, and a translation infrastructure. The system layer contains application functionality and so-called effectors and probes. Effectors are components that carry out system modifications and probes observe the application. The information collected by probes can be published and queried by the architecture layer. In the architecture layer information from the probes is aggregated and used to maintain an architectural model, which is periodically evaluated by a constraint evaluator. If a constraint violation occurs, an adaptation engine carries out corresponding actions via effectors. The translation infrastructure is responsible for transmitting information between the two layers.

The epi-aspects architecture and Rainbow framework both separate application functionality from an observation and adaptation mechanism. In the epi-aspects architecture, the observation and adaptation mechanism is the autopoietic system and in the Rainbow framework, this mechanism is the constraint evaluator in the architecture layer. A significant difference of both architectures is that an epi-aspect encapsulates the functionality of probes, effectors, and translation infrastructure, which are separate components in the Rainbow framework.

Adapta by Sallem et al. is a CORBA-based reflective middleware for developing adaptive, component-based applications. Like the Rainbow framework, and the epi-aspects architecture, it aims at separating application code from the code responsible for adaptation. Adapta has a runtime environment providing monitoring and trigger functionality. This environment can be configured by a XML-based reconfiguration language denoted as *AdaptaML*. This language allows developers to configure monitoring components, to define local and distributed events, and to specify reconfiguration actions that are applied to the application in response to events.

Adapta and the Rainbow framework focus primarily on adaptation, which is just one feature of the epi-aspects architecture. In comparison, the strength of the epi-aspects architecture is finer-grained error recovery, as epi-aspects can advise on exceptions and directly access the internal state of instances within the application.

7.2 Monitoring-Oriented Programming (MOP)

As described in [5], monitoring-oriented programming is a programming paradigm build upon runtime verification techniques that aims supporting reliable software via monitoring and error recovery. The formal specification of an application is used as the basis for generating a set of monitors that are integrated into the software. Chen et al. propose a concrete development Java development tool denoted as Java-MOP in [4], which provides user interfaces for editing and processing specifications for generating monitors.

Java-MOP is not directly comparable with the epi-aspects architecture, because it is essentially a compiler for monitor specifications. The epi-aspects architecture on the other hand is a runtime and development environment for conscientious software based on aspect-oriented programming. Chen et al. also propose an aspect-oriented approach for MOP in [5], in which the formal specifications are encapsulated in abstract aspects. Like Java-MOP this aspect-oriented MOP tool is not

dynamic, since the formal specifications are translated into monitors before runtime.

Like the epi-aspects framework, monitoring-oriented programming can be used to implement fine-grained error detection and recovery, and mechanisms for adapting the system. However, depending on the implementation MOP application are not necessarily dynamic and adaptive in a sense that a running system can be adjusted. It is possible to consider conscientious software and hence the epi-aspects framework as an extension of the MOP paradigm.

8. Discussion

This section discusses open issues and limitations of autopoietic conscientious software architecture, the proposed architecture, and the Epi-AJ framework that are not addressed in the other parts of the paper.

8.1 Limitations of Epi-Aspects

The proposed architecture encourages a clear separation between application functionality and an autopoietic system for monitoring, regulation, and error recovery. This architectural separation is a shift in software engineering practice, which focuses on application functionality and often neglects well-known error recovery and adaptation techniques. Since the autopoietic system is not an artificial intelligence, but implemented by developers who have designed rules for keeping an application running as smoothly as possible, certain unpredictable conditions can still cause the application to perform unwanted actions. Critical failures that crash the system can be handled by the autopoietic system. However, it is not possible to prevent an application from doing something it is not supposed to do. As such, the epi-aspects architecture is prone to human failure.

A practical issue of epi-aspects not addressed in the previous sections is the problem of potential buggy epi-aspects. Since epi-aspects can contain a significant amount of code, the introduction of latent bugs is possible. As a result, epi-aspects have to provide a mechanism that reliably performs self-updates. One possible approach is the usage of a "meta" epi-aspect that monitors the epi-aspects for internal problems.

Another issue is the update of epi-aspects. In the case study, we only implement a dedicated epi-aspect that provides a mechanism for updating the classes of the OIM system, but not the epi-aspects. The most straightforward approach for dealing with the issue is to implement a dedicated epi-aspect that provides functionality for reliably updating other epi-aspects and itself.

8.2 Realizing An Autopoietic system

The autopoietic simulator of the Epi-AJ framework is meant for development and test purposes. In order to use epi-aspects in real world applications, the development of a full autopoietic system is necessary. Apart from the lack of autopoietic programming languages as envisioned by Gabriel and Goldman in [9], the following issues have to be addressed.

The first question is how to implement and deploy an autopoietic system. One option is to implement is as a program that runs directly on the computer's hardware and provides a virtual machine for running an operating system, similar to VMWare or Colinux. The advantage of this approach is that the autopoietic system does not depend on other software, which might be buggy. Furthermore, components, drivers, and applications of the operating system can be realized as aspect oriented conscientious software that is woven into the autopoietic system on startup.

Another similar option is running the autopoietic system on top of an existing, stable operating system kernel, which provides hardware abstraction, basic services, and includes drivers.

A third option is to implement the autopoietic system as an application running on an operating system or inside a virtual machine. Advantages are that this approach has lower implementation complexity. The major disadvantage is that the autopoietic system depends on an operating system or virtual machine and therefore is only as stable as the underlying software.

Another technical issue that has to be resolved are the exact mechanisms for invoking recommendation and query advice woven into the autopoietic system, and for transporting messages from epi-aspects to the autopoietic system via an epi-queue. If autopoietic system and application run in the same process, which is the approach used by the autopoietic simulator, this issue is trivial. However, running the autopoietic system and application in the same process defeats the purpose of conscientious software, because a critical failure in the application might terminate the process and thus the autopoietic system.

9. Conclusion

This paper proposes a concrete aspect-oriented architecture for realizing conscientious software as envisioned by Gabriel and Goldman in [9]. Apart from proposing and describing the conceptual architecture, which introduces epi-aspects as a construct for combining an autopoietic system and applications into working conscientious software, we design and implement Epi-AJ, a framework for implementing and testing aspect-oriented conscientious applications. Furthermore, we conduct an experimental case study to evaluate the potential of the proposed architecture and Epi-AJ framework. The results of the case study show that in comparison with a plain Java application, an aspect-oriented conscientious application adapts better to changes and problems that we introduced in our controlled test environment.

The research presented in this paper is only the first step towards creating a solid aspect-oriented architecture for conscientious software that is suitable to be used in real-world applications. We expect further research to focus on two areas: One area is the design of an autopoietic programming language that can be used to implement the autopoietic system of the aspect-oriented conscientious software architecture. The second area is experimental and comparative research that continues to validates and compare the proposed architecture to other techniques for creating self-sustaining software, such as autonomic computing, reflective and adaptive middleware, and monitoring oriented programming.

Acknowledgments

We would like to thank our shepherd Doug Lea who provided guidance and helpful feedback on improving this paper. We would also like to thank Ron Goldman, and the anonymous reviewers for their comments and direction.

References

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotk, O. Lhotk, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc : An extensible aspectj compiler. *Transactions on AOSD*, (1):293–334, 2006.

[2] G. S. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the lancaster experience. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 262–267, New York, NY, USA, 2004. ACM Press.

[3] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[4] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[5] F. Chen and G. Roşu. Mop: Reliable software development using abstract aspects. Technical Report UIUCDCS-R-2006-2776, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[6] F. Eliassen, E. Gjørven, V. S. W. Eide, and J. A. Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 1, New York, NY, USA, 2006. ACM Press.

[7] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM Press.

[8] S. Fleissner and E. Baniassad. A commensalistic software system. In *OOPSLA '06: Companion to the 21st annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2006.

[9] R. P. Gabriel, R. Goldman, and K. A. McIntyre. Conscientious software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2006. ACM Press.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[11] P. Grace, G. Coulson, G. S. Blair, and B. Porter. A distributed architecture meta-model for self-managed middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 3, New York, NY, USA, 2006. ACM Press.

[12] P. Greenwood and L. Blair. Using dynamic aop to implement an autonomic system. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), Lancaster*, pages 76–88. RICAS, March 2006.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[14] R. Maia, R. Cerqueira, and F. Kon. A middleware for experimentation on dynamic adaptation. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM Press.

[15] B. McMullin. Computational autopoiesis: The original algorithm. Working Paper `97-01-001`, Santa Fe Institute, Santa Fe, NM 87501, USA, 1997.

[16] B. McMullin and F. J. Varela. Rediscovering computational autopoiesis. In *Fourth European Conference on Artificial Life (ECAL'97)*, pages 38–47, 1997.

[17] R. Murch. *Autonomic Computing*. IBM Press, March 2004.

[18] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[19] A. Rasche, W. Schult, and A. Polze. Self-adaptive multithreaded applications: a case for dynamic aspect weaving. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM Press.

[20] M. A. S. Sallem and F. J. da Silva e Silva. Adapta: a framework for dynamic reconfiguration of distributed applications. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 10, New York, NY, USA, 2006. ACM Press.

[21] F. J. Varela, H. R. Maturana, and R. Uribe. Autopoiesis: The organization of living systems, its characterization and a model. *BioSystems*, 5:187–196, 1974.

[22] M. Zeleny. Self-organization of living systems: A formal model of autopoiesis. *International Journal of General Systems*, 4:13–28, 1977.