Horizontal Decomposition of Prevayler

Irum Godil and Hans-Arno Jacobsen

Middleware Systems Research Group, University of Toronto

Abstract

Aspect-Oriented Programming (AOP) is an emerging paradigm to modularize crosscutting concerns. A series of guidelines to refactor a software system into a common core and a set of variable functionalities have become known as Horizontal Decomposition (HD). In this paper we apply the HD principles to the Prevayler main memory database management system. The objective is to evaluate and refine these principles and to extract patterns of their use through a case study on a popular open-source software system. Our evaluation shows that HD reveals six crosscutting functionalities. The refactoring of these concerns yield 36 different configurations of the Prevayler system which were previously not possible. The refactoring also reduces the core Prevayler code size by 53%, demonstrates a decrease of coupling between core functionality components by 43%, and reduces the lack of cohesion of the core system by 71%. Given the heterogeneous nature of crosscutting displayed in Prevayler, the size and separation of concern metrics have not reduced for the overall refactored system, i.e., for the core composed with the aspects. A posterior analysis of the re-engineering process reveals 22 refactoring patterns that could be readily used by an automatic aspect refactoring tool.

1 Introduction

A software system often supports a set of core functionalities. These functionalities constitute the essence of the system without which the system does not achieve its main purpose. For instance, the main goal of a database system could be seen as to persist data objects. Over the lifetime of the software system, it often evolves through the addition of new features and the modification of existing ones. While some of these changes are imperative for the core functional requirements addressed by the system, others mainly increase the capabilities of the system without being crucial to the main cause. The evolution and implementation of all these features most often results in entanglement in the code space of the system. Separation of concerns, feature modularization, and a configurable product line architecture of the system is thus not easily achievable.

Aspect-oriented programming (AOP), an emerging programming paradigm, aims at achieving increased separation of concerns and modularity, especially with respect to crosscutting features [1]. AOP's steady progress from "bleeding edge" research to mainstream technology [2], demands well-defined approaches, methodologies, and mechanisms for identifying and refactoring crosscutting concerns as aspects in software systems. The horizontal decomposition (HD) principles constitute one set of techniques for this purpose [3]. The HD principles are a set of guidelines to, firstly and most importantly, distinguish aspect functionalities from non-aspect ones, in order to lay out clear responsibilities for AOP and, secondly, to enable superimpositional architectures [3]. A superimpositional architecture is a software architecture that can be enriched with features represented by aspects according to the requirements of the user, the application domain, and the runtime environment. The HD principles have been successfully applied to resolve feature convolution from middleware systems [3]. Convolution in this context is a metaphor to refer to the interaction of many non-modularized features at a single point in the code space of the system. Con-

 In Proceedings of CASCON 2005, October 17-20, Richmond Hill, ON, Canada

volution prevents modularity and prevents the exclusion and inclusion of features as desired.

Our primary objective in this paper is to evaluate the applicability of horizontal decomposition in a domain other than middleware, where the principles originated. We perform this evaluation by applying the principles to the refactoring of a non-trivial, open source software system, by quantifying this refactoring effort through the use of standard software engineering metrics, and by reflecting on the refactoring process to gather universally applicable patterns that could guide other refactoring efforts. We selected the Prevayler database system for our purposes [4].

This paper makes three main contributions. First, we apply horizontal decomposition to the Prevayler database system validating the HD principles on a further case study. In the process we establish six features as crosscutting concerns and isolate them as configurable aspect components. The refactored aspect-oriented code for the system can be found at [5]. We also suggest certain modifications to make horizontal decomposition applicable to software systems exhibiting mostly fine granular crosscutting. Second, we quantify the results of performing aspect-oriented refactoring and empirically evaluate the benefits of horizontal decomposition. Third, we reflect on the refactoring process and generalize our efforts by discovering 22 refactoring patterns, out of which 17 are unique to our efforts and 5 patterns represent candidates independently developed in related work.

The rest of this paper is structured as follows. Section 2, discusses the Prevayler architecture, introduces the horizontal decomposition principles, and provides an overview of the aspect-oriented programming (AOP) paradigm. Section 3, presents a detailed study of the systematic application of the HD principles to the refactoring of the Prevayler system. To quantify the impact of applying horizontal decomposition, Section 4 presents software engineering metrics applied to the Prevayler sources before and after the refactoring. Section 5 reflects on the aspect-oriented refactoring process and extracts a list of refactoring patterns that crystallized in refactoring of Prevayler. Section 6 presents related work to put our efforts into perspective.

2 Background

2.1 Prevayler System Overview

Prevayler is a Java application that implements "Object Prevalence"; a concept developed by Klaus Wuestefeld and some colleagues at the Objective Solutions. The idea of Object Prevalence is to keep everything in RAM, as if "we were just using a programming language" [6].

Prevayler implements a fully-functional database in which a business object may be persisted. The business object must be serializable, i.e., implement the `java.io.Serializable` interface, and be deterministic, i.e., given an input the object's methods must always return the same output [6]. An architecture diagram of the Prevayler system is shown in Figure 1. A list of main Prevayler functionalities together with supporting components in the code space is also listed in Table 1.

The Prevalent Business Object represents the object to be persisted. Changes to the business object are made by executing Transactions[1] on it. A transaction maybe executed by itself, or may return a result after execution. A transaction that returns a result is known as a `TransactionWithQuery`. The business object maybe queried independently using the `Query` component.

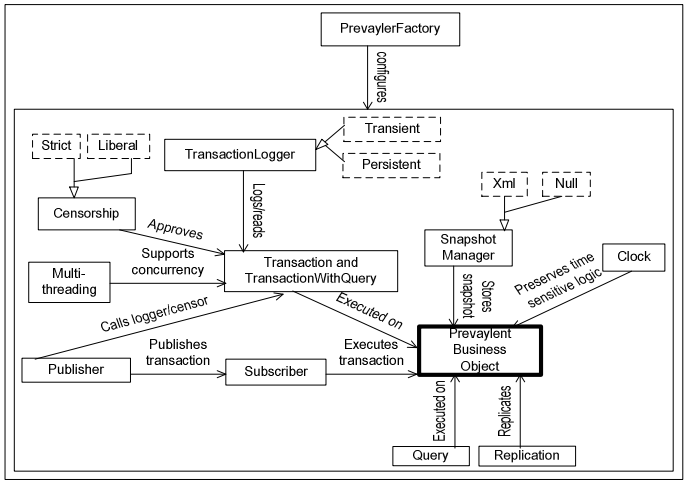

Figure 1: Prevayler System Architecture

[1] In this paper, we use the Arial Narrow font to distinguish Prevayler functionalities and `Courier New` font to highlight Prevayler components (i.e. classes and aspects).

Functionality	Classes in Prevayler Code Base
Prevayler Object	Prevayler, PrevaylerImpl, PrevaylerFactory
Transaction and Query	Transaction, TransactionWithQuery, Query, Prevayler, PrevaylerImpl, TransactionPublisher, CentralPublisher, AbstractPublisher, PrevaylerFactory, TransactionSubscriber, TransactionTimeStamp, TransactionWithQueryExecuter
Transaction Logging	TransactionLogger, PersistentLogger, DurableOutputStream, FileManager, SimpleInputStream, Stopwatch, TransientLogger, CentralPublisher, PrevaylerFactory
Snapshot Management	SnapshotManager, XMLSnapshotManager, NullSnapshotManager, PrevaylerFactory, PrevaylerImpl, Prevayler
Replication	ClientPublisher, TransactionPublisher, ServerConnection, TransactionSubscriber, POBOX, PrevaylerFactory, ServerListener
Multi-Threading	Turn, DurableOutputStream, PersistentLogger, TransientLogger, TransactionLogger, CentralPublisher, ServerConnection, ClientPublisher, PrevaylerImpl, BrokenClock, POBOX, AbstractPublisher, StrictTransactionCensor, ServerListener
Censoring	TransactionCensor, LiberalTransactionCensor, StrictTransactionCensor, PrevaylerFactory, CentralPublisher, SnapshotManager
Clock	Clock, PausableClock, MachineClock, BrokenClock, Prevayler, PrevaylerImpl, TransactionPublisher, CentralPublisher, AbstractPublisher, ClientPublisher, PrevaylerFactory, Transaction, TransactionWithQuery, Query, TransactionTimeStamp, TransactionWithQueryExecuter, TransactionSubscriber, POBOX, ServerConnection, TransactionCensor, LiberalTransactionCensor, StrictTransactionCensor, TransactionLogger, PersistentLogger, TransientLogger

Table 1: Prevayler Functionality and Components Summary

Prevayler achieves transaction durability via the Transaction Logging and the Snapshot Management features. Before a transaction is applied to the business object, it is logged by the `TransactionLogger`. Prevayler supports two kinds of logging mechanisms: Persistent Logging, which logs all the transactions to the file system and Transient Logging, which logs the transactions to an in-memory data structure. Additionally, in low-use periods, an object graph of the entire system, known as the snapshot, is stored on the hard disk. Snapshot Management is supported by various serialization mechanisms including Java and XML serialization methods. Upon failure recovery, the snapshot of the system is recovered first. Transactions in the log files that were applied after the snapshot was taken are then applied to the business object, resulting in a full system restore.

Execution of transactions to the prevalent object, failure recovery and logging of transactions is achieved by the `TransactionPublisher` and `TransactionSubscriber` components.

Prevayler supports rolling back of transactions through the Censorship feature. Before a transaction is logged, it is applied to a copy of the prevalent system. Only successful trials of transactions are eventually logged and executed on the business object [7]. Prevayler ensures that any transactions that are applied to the business object are time-sensitive; i.e., any future execution of the transaction will take the same time reference as it was first executed. This feature is very important when the exact time of the transaction is necessary for the business logic [8]. This feature is implemented by the `Clock` component.

Prevayler also implements Multi-Threading and Replication. With Replication, multiple clients can connect to a server, and the data on the server and all clients is always kept synchronized. Multi-Threading enables concurrent execution of the system.

Finally, the `PrevaylerFactory` configures the entire Prevayler system and all its components. Applications built on top of Prevayler define the business object to be persisted, and implement commands that are to be applied on the business object as Transactions.

2.2 Horizontal Decomposition

Horizontal decomposition (HD) is a set of principles that have been proposed in [3] to guide the aspect-oriented refactoring and implementation of complex software systems. We perceive that the implementation of an aspect consists of both its functional implementation and its possible interactions with every other aspect. HD promotes a two dimensional architecture in which the vertical architecture implements a minimum set of essential functionalities of the application as a *core* component, and the horizontal architecture captures crosscutting concerns including both functional and non-functional features [9],[10]. The

horizontal features are decoupled from each other and each can be independently woven into the vertical architecture. The core, also referred to as *primary functionality* is determined by primary requirements and the horizontal architecture is determined by aspect requirements. We defer a detailed presentation of each principle to Section 3, where we successively apply them to the Prevayler system.

2.3 Aspect Orientation

Aspects constitute crosscutting concerns [1]. Aspect-oriented programming allows the developer to cleanly encapsulate aspects in separate modules [1]. Examples of common aspects include security, access control, and error handling. Aspects are modularized by aspect language, such as AspectJ [11], Hyper/J [12], AspectC++ [13], and CME [14]. The primary decomposition of a system is implemented with a component language, such as Java or C++. AspectJ, an aspect-oriented extension to Java, is one of the most mature aspect languages today. AspectJ defines a set of new language constructs to support two kinds of crosscutting mechanisms: dynamic crosscutting and static crosscutting. Dynamic crosscutting is defined by means of join points that denote well-defined points in the execution of a (Java) program. A pointcut refers to a collection of join points and to parameters associated with the join points. A method-like construct, referred to as an advice, is used to define aspect code executed *before*, *after* or *in place* of a join point. *Inter-type declarations* are used to implement static crosscutting, which allows the developer to introduce new fields and methods into classes or interfaces to modify the existing type hierarchy. An aspect module includes pointcuts, the associated advices and possible inter-type declarations.

3 Horizontal Decomposition

In this section, we explain the application of the HD principles on the Prevayler system. Each section below is dedicated to one of the principles.

3.1 Relativity of Aspects

According to the first principle of HD, *the semantics of an aspect is determined by the primary functionality of the application*. The definition of object prevalence found in [6] states: "*In a prevalent system, everything is kept in RAM, as though you were just using a programming language. Serializable command objects manage all the necessary changes required for persistence. Queries are run against pure Java language objects.*" Thus, inspired from this definition, we define the primary functionality of Prevayler to be an in-memory database for business objects. Prevayler aspects are then all the features that crosscut this functionality in the code space.

3.2 Defining the Prevayler Core

The second principle of HD states that, *the basis of aspect-oriented decomposition is the establishment of a functionally coherent and vertically decomposed core*. Hence, based on the primary functionality defined above, the Prevayler core must support an in-memory database application for business objects. The Prevayler business object is central to the entire system, and thus all components that define and configure it must form part of the core. Additionally, in Prevayler, transactions are essential for making any updates to the business object. Since updating the business object is central to the proper working of the in-memory database, all components representing transaction and query objects are included in the core. To realize transactions, the core only supports transient logging; i.e., maintains the sequence of transaction logs in an in-memory data structure only, without persisting them to disk.

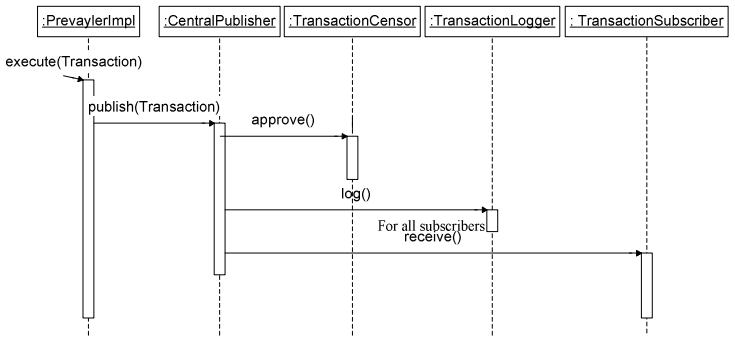

Figure 2: Execution of a Transaction in Prevayler

A sequence diagram representing the execution of a transaction is shown in Figure 2. Prior to logging transactions, the system censors transactions. As discussed in Section 2.1, Censorship ensures that a transaction execution works on a copy

of the business object, before applying it to the original system. We consider this to be an optional feature and exclude it from the core. However, we include in the core, parts of `TransactionPublisher` and `TransactionSubscriber` components that support publishing, logging and receiving of transactions.

Other features supported by Prevayler include replication between a server and multiple clients, support for time-sensitive business logic and execution of multi-threaded applications. None of these functionalities are essential for an in-memory database, and hence are excluded from the core.

3.3 Defining Prevayler Aspects

According to the third principle of HD, *using the core as a reference, a functionality is considered orthogonal if both its semantics and its implementations are not local to a single component of the core. Only the orthogonal functionality is treated in the aspect-oriented way.* In order to define crosscutting features in Prevayler, we propose the following modifications to the HD principles:

1. Since Prevayler is a rather small application, (i.e., 2418 lines of code and 60 classes), a feature is an aspect if it crosscuts through more than one class. Additionally, we consider a feature to be an aspect if its code is scattered through more than one method of the `PrevaylerFactory` class. This is mainly due to the small size of Prevayler and also because there is only one factory class which configures everything. For a larger system, there would be more than one factory class to configure different features of the system, which would cause more scattering.
2. The HD principles, only consider crosscutting in one system, and do not observe aspects that transcendent system boundaries. However, to also capture crosscutting for an end-to-end scenario involving the system, applications that are built on top of it, and the underlying operating system, we capture crosscutting in a system built on top of the Prevayler application. For this purpose, we consider the demo bank application, shipped with the Prevayler code. It uses Prevayler to persist bank transactions.
3. We have defined various levels at which a concern crosscuts through the system functionality. These levels are defined in Table 2. The purpose of such a classification is to know how scattered the concerns are across the code base. Note, from the crosscutting definitions, one concern can exhibit more than one crosscutting level.

Crosscutting	Concern Code Scattering
Intra-Method	>1 method of same class
Intra-Class	>1 class of same package
Intra-Package	>1 package of same system
Intra-System	Main code base and in application(s) built on top of code base

Table 2: Crosscutting Levels

We summarize below the Prevayler aspects with their functionality, logical independence from the core and crosscutting natures. Figure 3 shows the crosscutting of the Prevayler aspects. The various crosscutting levels inside Prevayler are summarized in Table 3.

Snapshot Management (S): Supports storing the entire object graph of the business object.
Logical Independence: The core only supports in-memory storage of the business object.
Crosscutting: Crosscuts through the `PrevaylerFactory` class which configures the type of Snapshot Management. Support is also provided in the `Prevayler` and `PrevaylerImpl` class for taking snapshots of the business object. In the bank application, the code crosscuts through the `Main`, `MainReplica` and `MainXML` classes for taking snapshots.

Censoring (C): Supports approving of transactions before being applied to the system, so that transactions that cause failures are not applied to the business object.
Logical Independence: The core supports simple transactions with no prior checking.
Crosscutting: The `PrevaylerFactory` class supports initialization and configuration of liberal or strict censoring. `CentralPublisher` has functionality to censor a transaction before it is published.

Replication Support (R): Supports duplicating contents between a server and multiple clients.
Logical Independence: The core supports only local publishing and receiving of transactions.
Crosscutting: In the Prevayler system, the replication concern is scattered through several meth ods in `PrevaylerFactory` only. In the bank

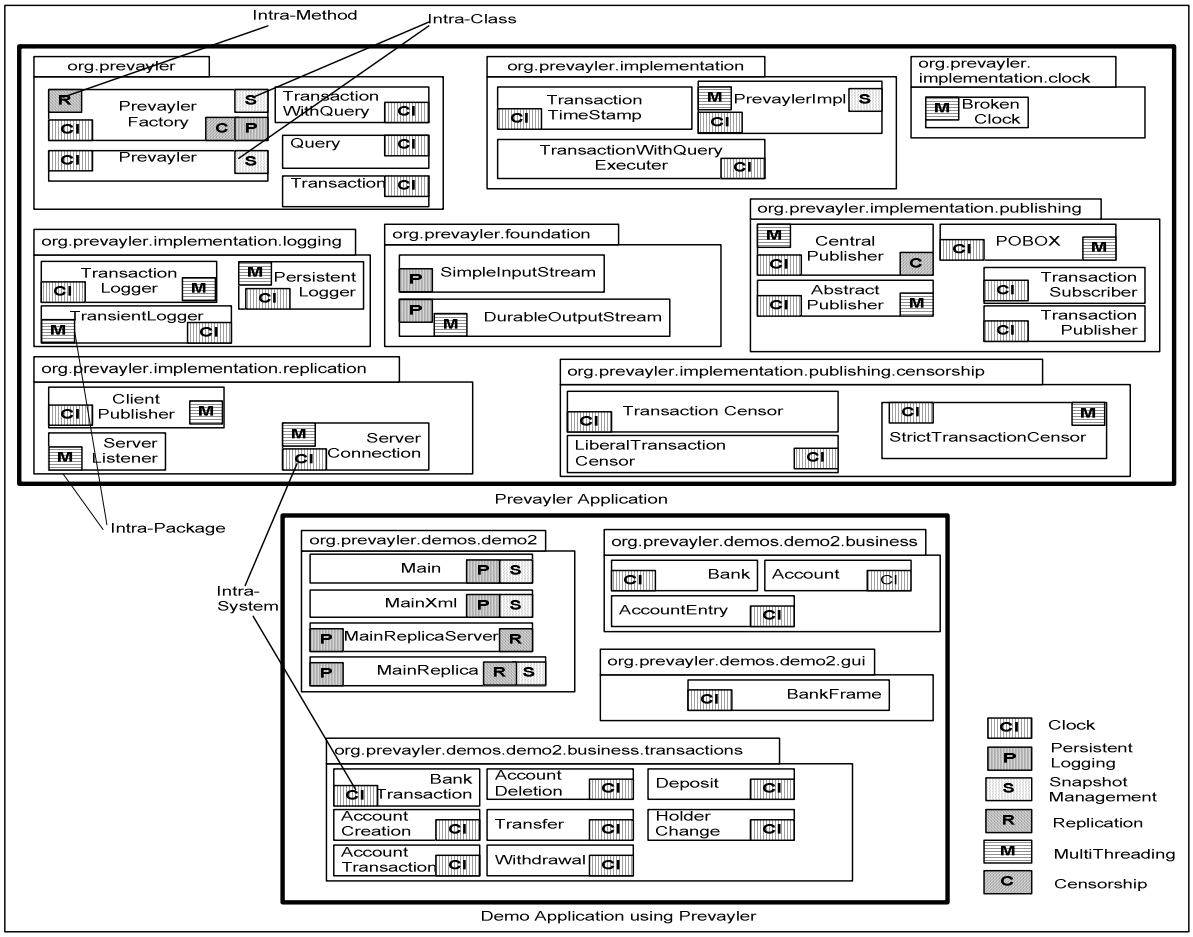

Figure 3: Aspect Crosscutting in Prevayler

application there is code to configure the replication client and server in `MainReplica` and `MainReplicaServer`.

Persistent Logging (P): Supports logging of transactions to the file system. Logging in Prevayler is different from the conventional logging feature used for debugging and other purposes; instead logging is an important functionality of Prevayler to support transaction durability.
Logical Independence: The core supports transient logging only.
Crosscutting: The classes `DurableoutputStream`, `SimpleInputStream` are used solely for persistent logging purposes. The `PrevaylerFactory` has variables to keep track of a transaction log file's age and size. It provides APIs to configure Persistent Logging in the system. By default the Prevayler system uses Persistent Logging. All sub-programs of the bank application also use Persistent Logging by default, except for the `MainTransient` class, which demonstrates Transient Logging.

Clock Support (Cl): Supports time-sensitive business logic, so that future executions of a transaction will take the same time reference as the first execution[2].
Logical Independence: The core does not support time-sensitive business logic.
Crosscutting: Crosscuts through `PrevaylerFactory` for clock configuration. `Prevayler` and `PrevaylerImpl` support determining of execution time of transactions and queries. Transaction and query classes support time-sensitive transactions. Local and remote publishing and subscribing, and logger and censor classes support

[2] Future executions of a transaction are necessary when restoring a system, mainly after a failure.

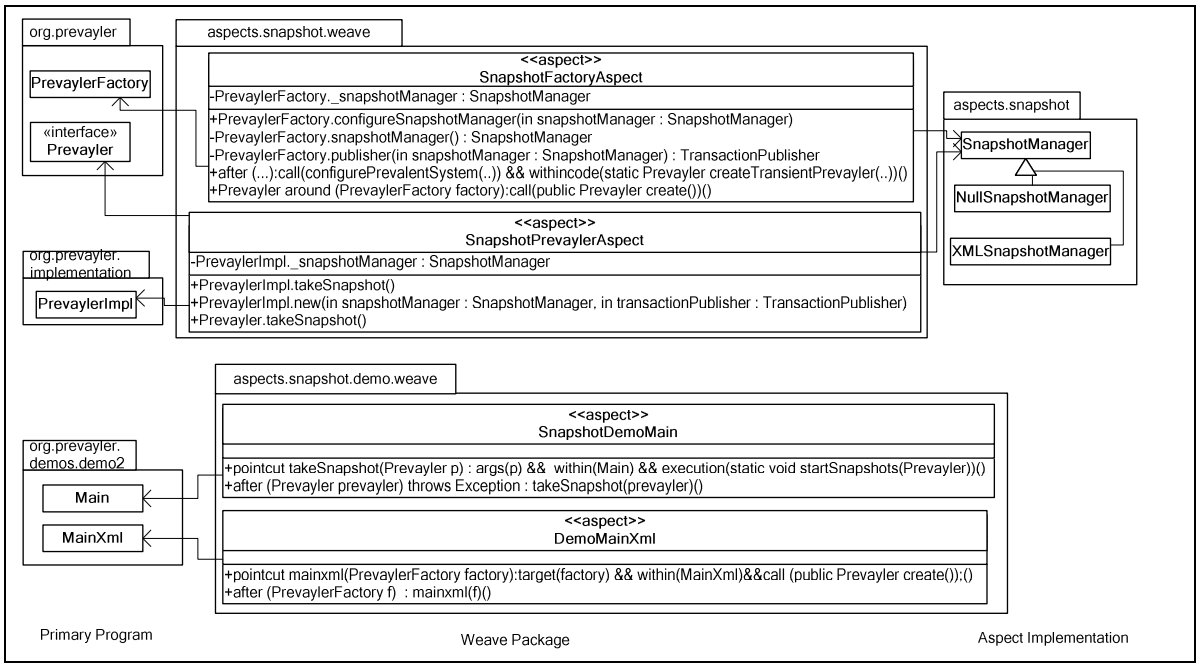

Figure 4: Aspect Implementation of Snapshot Management Concern

time-sensitive transactions. It also crosscuts through all classes of the bank application that support transactions on bank objects. Additionally, the `AccountEntry` and `BankFrame` class also support clock features.

Aspect	Intra-Method	Intra-Class	Intra-Package	Intra-System
S	X	X	X	X
C	X	-	X	-
R	X	-	-	X
P	X	-	X	X
Cl	X	X	X	X
M	X	X	X	-

Table 3: Aspect Crosscutting Levels in Prevayler

Multi-Threading (M): Supports application of transactions from multiple threads.
Logical Independence: The core supports single-threaded applications only.
Crosscutting: Multi-threading is scattered in form of synchronized methods, classes extending `java.lang.Thread` class, support for wait and notification, and sequencing operations to be performed in order by several threads in the logging and publishing layer, all the client/server classes that support replication, and the `PrevaylerImpl`, `StrictTransactionCensor`, `BrokenClock`, and `DurableOutputStream` classes.

3.4 Resolving Convolution

According to the fourth principle of HD, *crosscutting concerns should be implemented class-directional towards the core.* The goal of aspect-oriented treatment is to eliminate the convoluted implementation in the original Prevayler code base. This is accomplished through the implementation of orthogonal functionality as a set of core aspects of Prevayler and the untangling of the code convolution among aspects themselves. We present the details in the sections below.

3.4.1 Implementing Prevayler Aspects

The implementation of aspects generally consists of two distinct parts: the implementation of aspect functionality and the implementation of the interaction between the aspect and the core. The classes which solely support the main functionality of a crosscutting concern are moved into a new `aspects` package. The crosscutting code is then implemented as aspects inside a `weave` package. Furthermore, the crosscutting code in the bank applications is refactored into a `demo.weave` package.

Figure 4 shows the refactoring of the Snapshot concern. We have moved the core classes: `SnapshotManager`, `NullSnapshotManager` and `XmlSnapshotManager` into the `aspects.snapshot` package. We have introduced a weave package to implement the crosscutting functionality in aspects. The bank application's crosscutting code is placed inside the `demo.weave` package.

3.4.2 Untangling Convoluted Aspects

We observe the following two areas of convolution among aspects:

Replication with Snapshot Manager: We observe that the same code for replication support in the `PrevaylerFactory` will occur at different join points in the case of just the core code vs. the case when the Snapshot aspect is introduced. As a result, we introduce two different weave packages and aspects, i.e., `ReplicationAspect` which crosscuts through the refactored `PrevaylerFactory` code and the `ReplicationSnapshot` aspect which crosscuts through the advices inside `SnapshotFactoryAspect`. This is shown in Figure 5.

Clock and Multi-Threading (Cl&M): We see convolution between the clock and multithreading support in Prevayler. There are a lot of APIs in Prevayler where both Cl&M parameters are present; as a result we need to define APIs for 4 possible configurations. An example is shown for `TransactionLogger's` log function in Figure 6. This kind of convolution is also found in Persistent Logger and Censorship aspects. Furthermore, we see independent convolution with replication and of just multithreading convoluted with Snapshot Manager.

Table 4 summarizes the aspect convolution. The last row for Cl&M shows that we had to introduce an additional `weave` package for supporting both clock and multi-threading together in the core, which takes precedence over regular Cl&M advices over the same join points. In this row, where check marked, we had to add a `clock-multithreading` package for the aspects as well. For instance from column 1, we have aspects and weave packages to support censorship and additionally 3 packages: `censorclock.weave`, `censorthread.weave` and `censorclockthread.weave`.

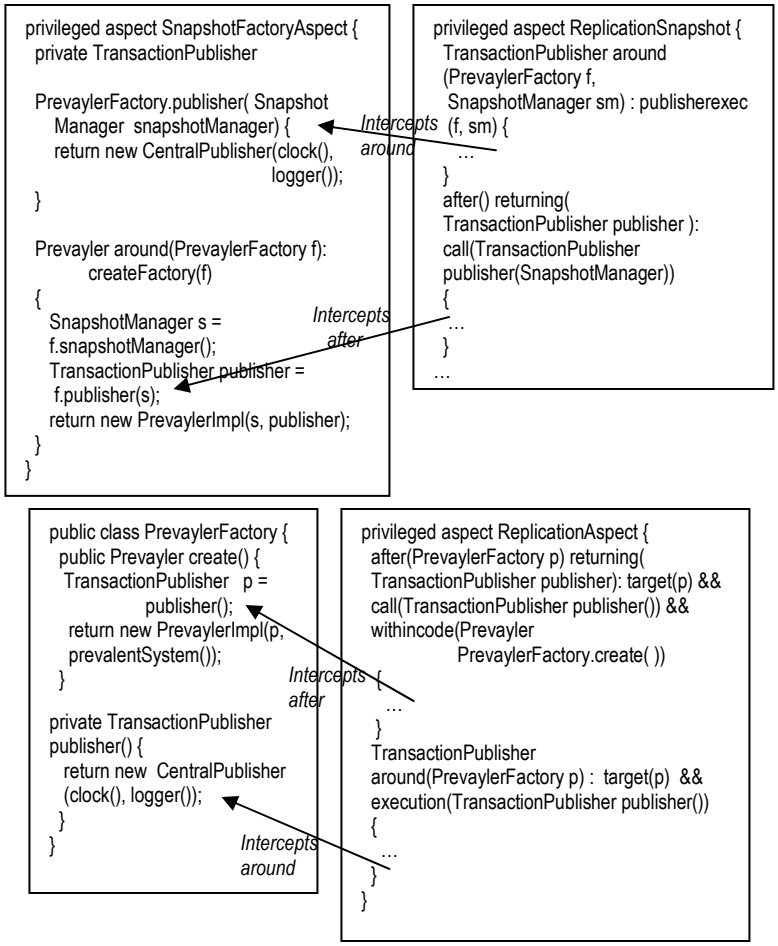

Figure 5: Replication Aspect Convolution

	C	P	R	S
R	-	-	-	X
M	X	X	X	X
Cl	X	X	X	-
Cl&M	X	X	-	-

Table 4: Aspect Convolution in Prevayler

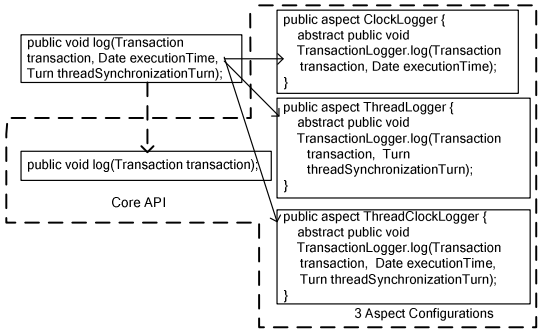

Figure 6: Cl&M Aspect Convolution

3.4.3 Aspect Dependencies

We discovered that not all our aspects are independent, i.e., we cannot configure a version of Prevayler with any one of them without support from another aspect. More precisely, we made the following observations:

Transaction Censoring: is not fully independent as Censoring uses Snapshot Manager to get a copy of the stored business object, and executes transaction on it first to test if it actually works.

Prevalence Base: `PrevaylerFactory` defines APIs and fields to configure the directory where snapshots and persistent logs are stored. We put this code into the `Prevalence-Base.weave` package. Snapshot Management and Persistent Logging aspects both depend on it.

Replication: Replication is difficult to support without multi-threading. There's one server and many clients and any of those can publish a transaction at any time. The server has one thread waiting for connections from the clients and upon each connection it makes a new thread to handle that connection. Thus, without threads it is difficult to handle the client connections.

We also observe that the core can be compiled independently without having any knowledge of the aspects.

3.5 Incremental Decomposition

The fifth principle of HD states that, *decomposition in the aspect dimension is assisted by incremental refactoring*. Our experience shows that the definition of the minimal core is adjusted and refined gradually over time. Consequently, we discover new aspects as the definition of our core refines. We did three stages of incremental refactoring to arrive at the six aspects mentioned. We list the summary of the process in Table 5. For instance, our initial refactoring (Stage 1) starts with aspects Snapshot Management(S), Censoring(C) and Replication(R), listed under column A, with the aspects in column B yet to be identified. In stage 1 we found one convolution of R on S, shown in column C. In Stage 2 discovery of the Persistent Logger(P) aspect, leads to removal of `PrevalenceBase` code from the core and as a result the Snapshot aspect needs modification. Finally, in Stage 3, the Clock(Cl) and Multi-Threading(M) aspects require modification of all other aspects together with the core.

Stage	A	B	C
1	S,C,R	P,Cl,M	R → S
2	P	Cl, M	P,S→PrevalenceBase
3	Cl, M	-	C, P, R, S

Table 5: Incremental Decomposition Summary

4 Evaluation and Metrics

We perform an empirical evaluation of the Prevayler code before and after the refactoring, as this is important to draw quantitative conclusions about usefulness of the procedures applied. In the sections to follow we present a brief explanation of these metrics, followed by the results and analysis on Prevayler. We conclude this section by evaluating usefulness of AOP and HD principles in the light of these metrics. A detailed definition of the metrics can be found in [15].

4.1 Metrics Suite

We have categorized our analysis into the following two metric suites:

Separation of Concerns (SoC): We measure SoC using Concern Diffusion over Components (CDC) and Concern Diffusion over Operations (CDO) [15],[16]. CDC counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them. CDO counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.

Coupling, Cohesion and Size: Coupling counts the number of other classes and aspects to which a class or an aspect is coupled. This metric is useful in evaluating the correlation between various components of the system and to measure dependencies between classes [15]. Cohesion measures the lack of cohesion of a class or an aspect. It measures the dissimilarity of methods in a class by its attributes [17].

We used various metrics mentioned in [15] for calculating size. These include: Vocabulary Size (VS), which counts the total number of components, i.e., classes and aspects in a system; Lines of Code (LOC) of the system; the Number of Attributes (NOA) relating to a specific concern and Weighted Operations per Component (WOC),

which measures the complexity of a component in terms of its operations.

4.2 Assessment Procedures

The SoC metrics, VS and NOA were calculated manually for the code before and after refactoring. The rest of the metrics were gathered using the CASE tool Together [17]. The specific tool metrics that we used are: CBO [3] for coupling, LCOM3[4] for cohesion, LOC for lines of code and WOC for weighted operations per component.

The values for these metrics for each class in the original system and the refactored system were summed up to get the total metric, respectively. We have classified the refactored system's metrics into metrics just for the core system and for metrics for the core together with classes refactored into the `aspects` packages for the system. Since there is no tool support for counting data for aspects, we have not measured metrics for aspect files, except for the LOC metric which we measured by writing our own scripts.

4.3 Results and Analysis

Based on the metric definitions above, we expect that after refactoring, all numbers will reduce for the core code vs. the original system. We do not necessarily expect the metrics to reduce for the entire refactored system including aspects; as we observe that these numbers depend on the nature of the crosscutting code. Homogenous crosscutting code would result in reduction of size and SoC numbers; whereas no prior claims can be made for heterogeneous crosscutting code. Following are the results and analysis of the application of these metrics.

Separation of Concerns: The CDC and CDO values before and after the refactoring are shown in Figure 7. From our refactoring process, there is no reduction in the number of components which support the main functionality, as they are moved into `aspects` packages as they are.

Furthermore, we observe that the crosscutting logic in Prevayler, unlike Logging or Tracing[5] functionality, is not a common piece of code that is scattered over multiple components. Instead, crosscutting is found in the form of instance variables specific to components, or parts of method bodies that are dedicated to configuring or invoking a concern code. As a result, the crosscutting code from all components cannot be grouped into one aspect. Thus, to this end, for each class that displays crosscutting logic in the core, we introduce one aspect for each concern; and move the crosscutting code into the aspect. In some rare cases, we have concentrated more than one class' crosscutting code in an aspect.

Thus, the results of CDC where the OO and AO numbers are equivalent are not surprising. The Clock(Cl) concern is worth noting where the AO numbers are less than the OO numbers. We observe that this is due to the existence of eight transaction classes in the bank application, whose code has been concentrated into one aspect; due to commonality of code. Next, we observe 3 cases, where the CDC numbers for the AO version are greater than the OO version. Replication(R) has one extra `ReplicationSnapshot` aspect for the aspect convolution discussed in Section 3.4.2. Persistence Logging(P) has 2 extra components, which are accounted for by the `Prevalance-Base` aspects. Multi-threading(M) has 8 extra components. Seven of these are due to the clock-thread convolution as explained in Section 3.4.2, and one is an abstract aspect for synchronized keyword refactoring. Note, that although Snapshot Manager(S) also uses the `PrevaylenceBase` aspect, but we have concentrated 2 cases of similar code into the same aspect; due to which CDC does not increase.

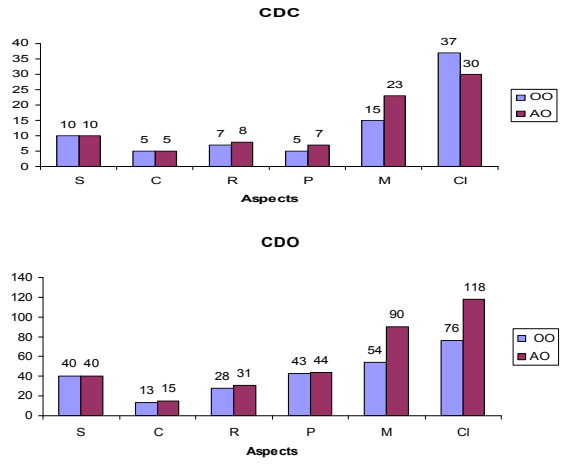

Figure 7: SoC Metric Values

[3] CBO is Coupling Between Objects metric in [17]

[4] LCOM3 is Lack Of Cohesion Of Methods 3 in [17]

[5] We refer to conventional Logging and Tracing features used for debugging and other purposes.

For CDO, we observe a steady pattern of higher AO numbers vs. OO except in case of (S). For (P), this is due to the `PrevalenceBase` aspect; again for (S) some homogenous code is placed in the same advice which counterbalances the `PrevalenceBase` code and does not increase CDO. For (R) we have extra advices in the `ReplicationSnapshot` aspect, and for (C) we have a helper function in the code base to capture a pointcut and an auxiliary advice for a crosscutting point. The difference for (M) and (Cl) are much higher, i.e., an increase of 66% and 55% in the AO version. This difference is mainly due to existence of the four configurations of multithreading and clock as mentioned in Section 3.4.2 and due to presence of auxiliary advices to capture local variables and parameters.

Coupling, Cohesion and Size:
The metric charts before and after the refactoring are shown in Figure 8. We notice a reduction in coupling and lack of cohesion, from the original code base in both the core code and core with aspect classes. For the core case, the coupling is reduced by 43% and lack of cohesion by 71%. These are very optimistic numbers, and conform to our understanding that removal of crosscutting functionality should make the core less-tightly coupled and more cohesive. For the size metrics, we see a decrease in the Vocabulary Size (VS) for the core code from the original. This affirms our hypothesis that application of HD principles makes the core compact, as the core is meant to concentrate only on the key functionality of the system. We notice that the VS numbers are higher for the core with aspects as compared to the original code base. The reason is exactly as that mentioned for CDC numbers.

The LOC numbers have decreased by 53% from the original code vs. the refactored core. This is again a positive result. The numbers have however, increased for the AOP code. The reason is similar to that of CDC numbers. Due to non-homogenous crosscutting code, we do not save on crosscutting code lines. Instead, the creation of aspects and pointcuts only increase this number.

For WOC, we see a decrease of 40% from the core versus the original code and a slight decrease of 0.2% from AO vs. the OO code base. The numbers are optimistic for the core case. With the removal of additional functionality from the system, the reduction of these numbers validates the accuracy of our work.

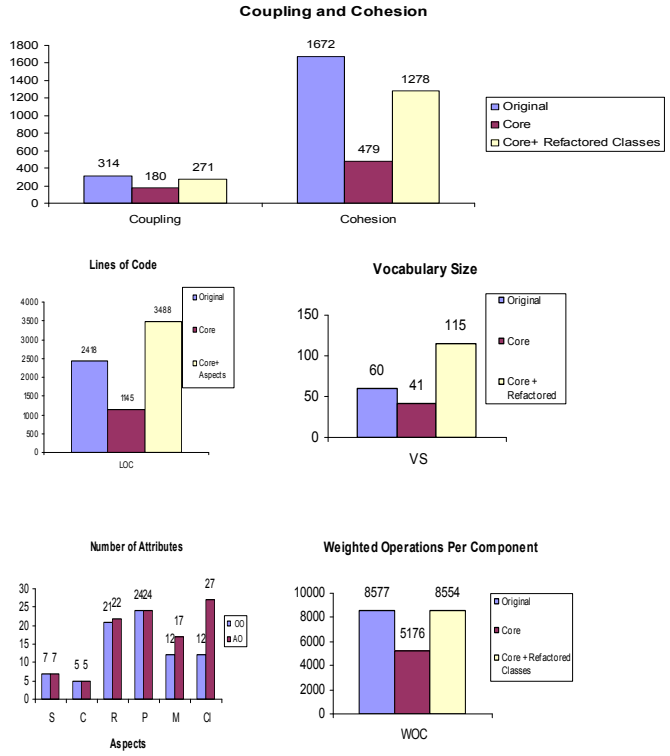

Figure 8: Coupling, Cohesion and Size Metrics

For the number of attributes, we see 3 cases of equal values for AO and OO version, and 3 cases where the AO numbers are greater than the OO numbers. The increase in all cases is due to the presence of helper variables that need to be introduced, for each local method variable. This number is significantly higher in the (Cl) aspect as compared to others.

4.4 AOP and HD Evaluation

Based on the metric results above, we draw conclusions on the effectiveness of AOP and HD principles as follows.

First of all, we notice that the application of the HD principles reduces the overall size of the core and produces a more cohesive and less-tightly coupled core. This shows that HD principles have been successful in eliminating secondary functionality from the core and have made the core code more compact and concentrated towards the key functionality. This is the goal of

the HD principles and has been achieved successfully.

Secondly, we see that the presence of aspects have increased the vocabulary size, CDC and CDO in most cases. As mentioned previously, we did not expect these numbers to necessarily decrease, given the heterogeneous nature of the crosscutting concerns. Although these numbers have increased, we have still achieved higher code modularity by separating crosscutting functionality into aspect packages and cleaning the core code from convolution.

As shown in Figure 9, we have created 36 configurations of the system, based on combinations of crosscutting features, which were previously not possible.

Thus, a simpler and more compact core code base, a large number of configurations of the database, and the separation of code crosscutting concerns into modular aspect units show that both AOP and HD principles have been successful in their application.

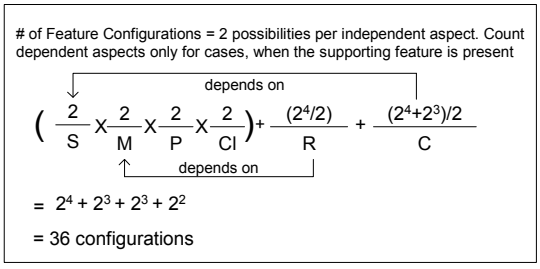

Figure 9: Number of Prevayler Configurations

5 AOP Refactoring Patterns

During our refactoring work on the Prevayler system, we extracted refactoring patterns that can be used for aspect-oriented refactoring on any system. Our focus was specifically on aspect-oriented refactoring patterns and not on object-oriented refactoring patterns. We discovered a total of 22 patterns out of which 17 are unique to our work and 5 have already been mentioned in [18],[19].

In this section, we present an overview of these patterns. All the unique refactoring patterns are listed in Table 6. We present the details of some of the non-trivial unique refactorings below, in the same format as [19] and Fowler *et al.* [20], however due to space limitations, we omit mechanics and present examples only for a few interesting cases. Finally, Table 7 mentions the validation of our refactorings from [19]. Besides each refactoring name, we list its number and page from the catalogue. Also, in some cases we discovered some changes about these refactorings, which have been listed next to it.

Pattern 1: Initialize Abstract Class Fields
Typical Situation: The constructor of an abstract class is refactored into an aspect; and there exist non-refactored instance variables that are initialized at the time of the declaration.
Recommended Action: Create a constructor with no parameters in the original class, and copy the initialization statements of fields that are initialized at declaration time. Explicitly invoke the class constructor from the refactored aspect constructor.
Motivation: When the constructor of an abstract class is refactored into an aspect, the fields of the abstract class that are initialized at the time of declaration are not initialized by default. Such fields need to be initialized explicitly inside the constructor.

Pattern 2: Capture Local Variables
Typical Situation: Refactored code references a non-refactored variable, local to a method, in the original class.
Recommended Action: Declare the same variable inside the aspect. If the variable is being set by a piece of code that can be captured using a pointcut, then set the variable inside an advice around that pointcut. If the variable setting code cannot be captured by a pointcut, then extract the variable setting code into a helper method; and then create an advice for the helper method to capture the variable value.
Motivation: Often refactoring requires moving a part of a method body into aspect advice; which uses a variable declared and initialized inside the method. AspectJ does not allow capturing pointcuts around setting of variables local to a method. If possible, we can capture the variable value from the pointcut of the code where the variable is set. However, in some cases this is not possible; e.g., setting of a variable is done through a method call and there is more than one call to the same method inside the method that is to be refactored. Thus, we extract the variable initialization code to a helper function and then capture the variable by creating advice for the helper method.
Example: An example is shown in Figure 10. We want to capture the `transactionCandidate`

variable, but we cannot do so using a pointcut around the call to `readObject()`, since this call exists twice in the method body. We extract the setting point into a helper method, and then create an advice around this method call.

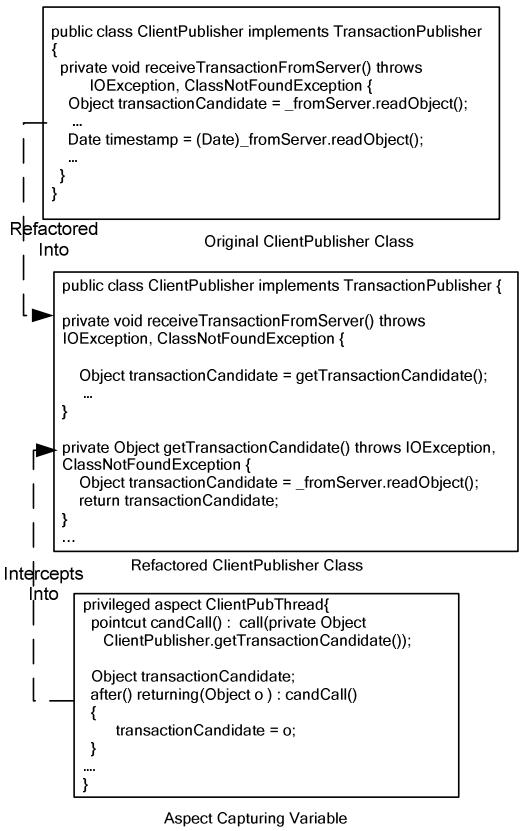

Figure 10: Capturing Local Variables Scenario

Pattern 3: Refactor Anonymous Class
Typical Situation: A method of an anonymous class that extends an interface is refactored such that the signature in the core changes and the original is moved to an aspect.
Recommended Action: Turn the anonymous class into an inner class; and add the refactored method as an inter-type declaration in an aspect.
Motivation: When the signature of a method in an interface changes, such that we remove a parameter relating to a concern; and there is an anonymous class implementation of the interface, the refactored method cannot be added as an aspect declaration. This is because, we want to keep the new method without some parameters in the core, and move the original into an aspect. We can refactor the anonymous class' implementation by "around-advice", but the original implementation must also remain in the core without the extra parameter. When woven with the aspect, the compiler complains that the old anonymous class must implement the refactored method.

Pattern 4: Refactor Inner Class Methods
Typical Situation: Inner class code that accesses variables of the enclosing class is refactored.
Recommended Action: Keep an enclosing class instance in the aspect, and initialize it through aspect advice. Use this variable inside crosscutting code of inner class in the advice.
Motivation: When inner classes use variables of the enclosing class, we cannot get the enclosing class' instance by using the "target" or "this" pointcut. Thus, to access the inner-class' methods or fields, we need to keep a separate local variable.

Pattern 5: Create Call-Within Pointcut
Typical Situation: Create a pointcut consisting of a method call that is within the context of another method body.
Recommended Action: Create two pointcuts one for the "call" of the method that is being intercepted and one for the "withincode" for the method within which the call exists. Note, the args() works for the "call" pointcut, but if we need to capture args() of the "withincode" pointcut then we have to do so by creating variables local to the aspect and capturing them in helper advices. Eventually, create an advice composed of the 2 pointcuts and crosscutting code.
Motivation: In Prevayler we saw many scenarios where the pointcut consisted of a method call that was inside of a method body.

Pattern 6: Remove a Method Feature
Typical Situation: A method implements crosscutting logic by using a parameter of a crosscutting type. The crosscutting parameter is used in only a few lines of the method body. We want to keep the method signature and most of its functionality in the core, while refactoring the part related to the crosscutting parameter.
Recommended Action: Remove the parameter from the method in the actual class, and move the original method signature into an aspect. Call the core method from the aspect method. Create advice to weave in the crosscutting code.
Motivation: In Prevayler we saw lot of cases where a method was refactored by removing one or more parameters of the method. We kept the

reduced method in the core and moved the method with all the original parameters into an aspect. The refactored method uses the same logic as the original code, except a few changes which we overcome by crosscutting advices.

Example: Figure 11 shows an example of refactoring the clock functionality from the `TransientLogger.log` function. The `log` function with the `Date` parameter is moved into the `ClockTransientLogger` aspect, which calls into the core `log` function. The "around" advice intercepts the core-functionality to introduce clock-logic in the method body. To simplify the illustration, we have omitted the implementation of multi-threading functionality in this example.

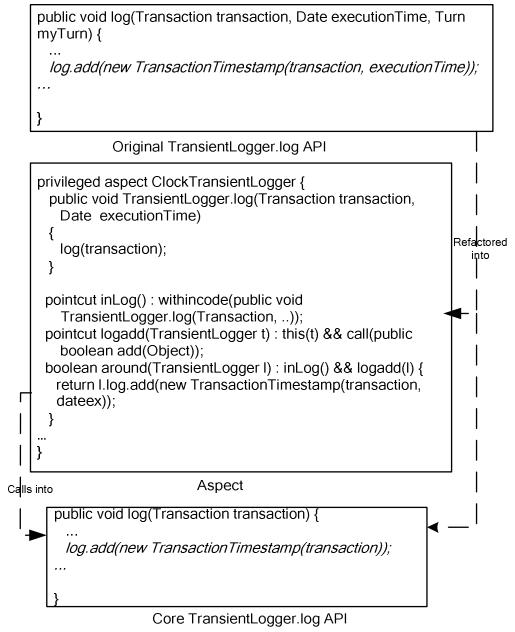

Figure 11: Removing a Method Feature

Pattern 7: Refactor Synchronized Methods

Typical Situation: Code contains a "synchronized" method and multi-threading is crosscutting.

Recommended Action: Create a marker interface for synchronized, and make target types implement the interface. Declare an abstract pointcut for synchronized method calls and create advice to synchronize the target object. Concretize the pointcut in specific aspects. We used the Generalize Target Type with Marker Interface refactoring ([19], 2.2.2) here.

Motivation: In Prevayler multi-threading refactoring consisted of removing synchronized keywords from method signatures. This refactoring is also applicable to other similar keyword refactorings.

Example: An example is shown in Figure 12. The `TransientLogger.update` method is refactored using the abstract and concrete aspects.

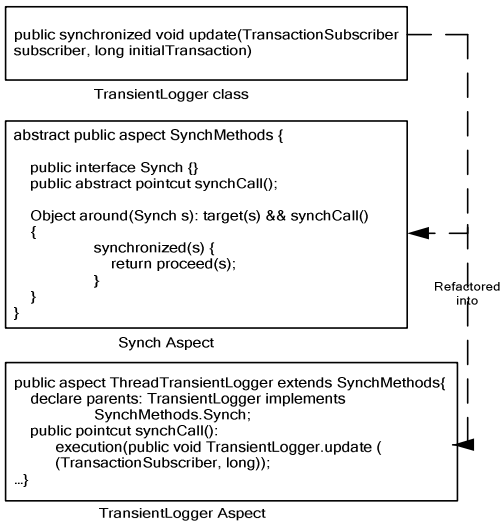

Figure 12: Refactoring Synchronized Methods

Pattern 8: Remove Catch-Finally Block

Typical Situation: Code inside a catch/finally block is only applicable to a crosscutting concern.

Recommended Action: If the try-catch/finally block is around the entire method body, refactor code into an around advice. But, if the block is around parts of a method, then extract the part enclosed by the try-catch/finally block into a helper method and then add an around advice.

Motivation: We noticed in Prevayler a lot of places where code relating to the multi-threading concern, was enclosed by a try-finally statement. This refactoring can be applied in similar cases and cases when the "catch" block catches exceptions relating to a crosscutting concern.

Pattern 9: Interface-Class Refactoring

Typical Situation: A method is protected in an interface but is public in its implementing class. Both the interface signature and implemented method are refactored into inter-type declaration.

Recommended Action: Make the interface signature and the implemented method in the inter-type declaration public. Sometimes, this may be overcome by introducing aspect protection ([19], 2.2.3). Special cases arise when an interface signature does not have an accessibility modifier.

Name	Typical Situation	Recommended Action
Create Aspect for Concern	Want to move crosscutting features from a class into an aspect.	Generate an aspect in a specific package for adding functionality related to the concern.
Increase Component Visibility	An aspect in a different package references a field or method of a private/ protected component.	Increase the class/interface visibility to public. We understand this refactoring reduces encapsulation, but we see no other alternative here.
Extract Interface Signature	Want to extract an interface signature as an inter-type declaration.	Move the method into an aspect as an inter-type declaration, and declare it abstract.
Introduce Helper Methods	The join point cannot be captured through any suitable pointcut.	Extract the crosscutting code into a new method ([20], pg. 110). Refactor crosscutting code using aspect advices.
Throw Soft Exceptions	An exception is thrown from an advice body, but not at its pointcut.	Throw a wrapped `org.aspectj.lang.SoftException` at this point.
Increase Child Method Visibility	Refactoring results in a child class method of low visibility to resemble a parent's method of higher visibility.	Give the child method same visibility as the parent's method. This may result in an increased visibility, but we cannot avoid it.
Extract Super Calls	Refactoring results into a super method call from an advice.	Extract ([20], pg. 110) the super call into a helper inter-type declaration.
Extract Final Fields	A final field, initialized in the constructor is to be made an inter-type declaration.	Remove final keyword from declaration. Add "declare error" to prevent setting of the field except in the constructor. If the field is set in an advice, extract setting code into a helper method, and exclude it from declare error.
Capture Local Variables	Refactored code references a non-refactored variable, local to a method.	Declare the same variable in the aspect. Set the variable through appropriate pointcuts and advices.
Initialize Abstract Class Fields	The constructor of an abstract class is refactored into an aspect; and non-refactored fields are initialized outside of the constructor.	Copy initialization statements into the refactored constructor. If the field is final, extract setting of field into a helper constructor; and call this constructor from the aspect.
Refactor Anonymous Class	Anonymous class' method, which extends an interface, is refactored. Signature in the core changes and the original is moved to an aspect.	Turn the anonymous class into an inner-class; and add the refactored method as an inter-type declaration in an aspect.
Refactor Inner Class Methods	Inner-class code that accesses variables of enclosing class is refactored.	Keep an enclosing class instance in the aspect, and use it inside crosscutting code of inner class.
Interface-Class Refactoring	Protected interface signature and its public implementation are changed to inter-type.	Make the interface signature and the implemented method in the inter-type declaration public.
Remove Catch-Finally Block	Code inside a catch/finally block is only applicable to a crosscutting concern.	Based on the implementation, either refactor the block as an around-advice around the method or extract code into a helper method and then create the around advice.
Refactor Synchronized Methods	Code contains a "synchronized" method and multi-threading is crosscutting.	Create marker interface and make target types implement it. Create abstract pointcut for synchronized method calls and a concrete advice. Concretize pointcut in specific aspects.
Remove a Method Feature	A method implements crosscutting logic by using a parameter of the crosscutting type.	Remove the parameter from the method in the actual class, and move the original method signature into an aspect. Introduce crosscutting logic via appropriate advices.
Create Call-Within Pointcut	Create a pointcut consisting of a method call that is within the context of a method body.	Create two pointcuts one for the *call* of the method and another for *withincode* for the method where the call exists. Compose crosscutting code and pointcuts into an advice.

Table 6: Summary of New AOP Refactorings

Name	Catalogue Reference	Our Modifications
Inline Class within Aspect	2.1.6, pg. 15	When the class does not use any non-public features of other classes in the same package, we move the entire class into the aspect package instead of inlining it.
Move Field from Class to Inter-type	2.1.8, pg. 17	We have introduced a new refactoring for final fields.
Move Method from Class to Inter-type	2.1.9, pg. 19	No modifications were made here.
Introduce Aspect Protection	2.2.3, pg. 26	[19] mentions making members visible to the child hierarchy of the aspect/class. Additionally, the field may need to be visible to other classes in the same package.
Encapsulate Implements with Declare Parents	2.1.2, pg. 5	No modifications were made here.

Table 7: Validation of Pre-existing AOP Refactorings

Motivation: Java allows protected interface signatures and public implementation methods, but AspectJ throws a run-time error.

6 Related Work

Refactoring of transactions and persistence has been the focus of various papers [21], [22], [23], [24]. Our focus has been on refactoring an existing code base, and not on developing a new application from scratch with aspects. Our refactoring techniques are similar; and like the others we use AspectJ to implement the aspects. The focus of [21], [22], [23] has been to aspectize transactions and persistence only. Our work goes beyond refactoring of transactions including a large number of other crosscutting concerns found in a database system. Although in [24], crosscutting concerns are identified, the process of doing so is through manual investigation only, and without using any standard mechanisms. We employ and illustrate a set of well-defined principles for this purpose. Driver *et al*. [24] focus on studying the effects of AOP on evolvability of a system. The others too present their results on usefulness of AspectJ to aspectize persistence and transactions. However, all studies lack an empirical evaluation to draw conclusions. We have focused intensively on using quantitative data to present our results.

In [3], the principles of HD have been developed and applied to the CORBA middleware implementation. We have applied the HD principles to Prevayler in a similar fashion. One of our objectives has been to validate the HD principles and study their effect in a different context. Moreover, we have extended the HD principles to identify crosscutting concerns at a different level of granularity and to account for concerns that extend into the application code layered on top of the system under investigation.

Fowler *et al*. [20] and [19] have presented code smells, i.e., symptoms that something may be wrong in the code, compounded by a catalog of refactorings through which these smells may be removed from the code. Although [20] focuses on improving the design of object-oriented systems; [19] like our work is geared towards refactoring Java applications by introducing AOP constructs. Our approach resembles that of [19]; in that we aim at gaining refactoring insights through experiments based on case studies. The refactorings mentioned in this paper extend the set of *Refactorings for Feature Extraction* presented in [19]. The refactorings presented in [19], are at a higher level of abstraction, whereas ours are more generic.

In [25], Cole and Borba present refactoring laws that can be used to derive or create behaviour preserving transformations. We can already see how some of our refactorings can be derived from these laws. For instance, the refactoring "Create Call-Within Pointcut" can be straight forwardly derived from the "Add before-call" and "Add after-call" laws. A formal derivation of all of our refactorings from the proposed laws is regarded as future work.

A quantitative study on modularization of design patterns with aspects is done in [16]. We have used similar metrics for our empirical evaluations. [16] presents the results for the OO and AO system only; whereas we have further broken down our AO results to also account for the core separately. In [16], a lower metric value in the AO version vs. the OO system is considered to be better. As mentioned in Section 4.3, given the heterogeneous nature of Prevayler's crosscutting concerns, and the fact that we create one aspect for each class that displays crosscutting code; this hypothesis is not necessarily valid for our study.

7 Conclusion

This paper presents our experience in applying the Horizontal Decomposition Principles to the Prevayler database system. Our overall goal was to validate horizontal decomposition in a context different from the context the principles were first conceived in. We have been successful in achieving this goal as manifest by our quantitative evaluations and the resulting configurability options for Prevayler. The refactoring of six crosscutting features has led to 36 different configurations of Prevayler. This was previously not possible and enables the creation of a product line out of Prevayler. Furthermore, we have removed implementation convolution from the Prevayler code base creating a cleaner and more compact Prevayler core. A reduction in coupling and lack of cohesion of 43% and 71% resulted, respectively. This shows that the refactored core is less-tightly coupled and highly cohesive compared to the original system. Our efforts have also

lead to discover aspect-oriented refactoring patterns. We have discovered 17 unique patterns and validated 5 pre-existing ones. This is a positive step towards standardizing AOP refactoring mechanisms. In the future, we aim to compose these patterns into a tool, thereby automating AOP refactoring.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-oriented Programming. In *Proc. Of the 11th European Conference on Object-Oriented Programming*, pages 220-242, Jyvskyl, Finland, 1997.

[2] D. Sabbah. Aspects – from Promise to Reality. In *Proc. Of the 3rd international conference on AOSD*, pages 1-2, Lancaster, UK, 2004.

[3] C. Zhang and H. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proc. Of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 188-205, Vancouver, Canada, 2004.

[4] Prevayler. http://www.prevayler.org

[5] Refactored Prevayler. http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem

[6] C. Villela. An introduction to object prevalence. http://www-106.ibm.com/developerworks/web/library/wa-objprev/index.html, 2002

[7] Prevayler – How Rollback Works. http://prevayler.codehaus.org/How+Rollback+Works

[8] J. Roubieu. Prevayler Tutorial. http://www.prevayler.org/upload/julien/prevayler_tutorial_en.rtf

[9] C. Zhang, Dapeng Gao and H. Jacobsen. Towards Just-in-time Middleware Architectures. In *Proc Of the 4th International Conference on AOSD*, pages 63-74, Chicago, USA, 2005.

[10] C. Zhang, Dapeng Gao and H. Jacobsen. Generic Middleware Substrate through Modelware. In *Proc Of the 6th International Middleware Conference*, Grenoble, France, 2005.

[11] AspectJ. http://www.eclipse.org/AspectJ

[12] Hyper/J. http://alphaworks.ibm.com/tech/hyperj

[13] AspectC++. http://www.aspect.org

[14] Concern Manipulation Environment. www.research.ibm.com/cme/

[15] C. Sant'Anna, A.Garcia, C. Chavez, C. Lucena, A. Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proc. Of the Brazilian Symposium on Software Engineering*, pages 19-34, Manaus, Brazil, 2003.

[16] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proc. Of the 4th International Conference on AOSD*, pages 3-14, Chicago, USA, 2005.

[17] Together Technologies. http://www.borland.com/together/.

[18] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Of the 4th International Conference on AOSD*, pages 111-122, Chicago, USA, 2005.

[19] M. Monteiro. *Catalogue of Refactorings for AspectJ*. Technical Report UM-DI-GECSD-200402, Universidade do Minho, 2004.

[20] M. Fowler (with contributions by K. Beck, W. Opdyke and D. Roberts). *Refactoring Improving the Design of Existing Code*. Addison Wesley, 2000.

[21] S. Soares, E. Laureano and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. Of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 174-190, Seattle, USA, 2002.

[22] A. Rashid and R. Chitchyan. Persistence as an Aspect. In *Proc. Of the 2nd international conference on AOSD*, pages 120-129, Boston, USA, 2003.

[23] J. Kienzle and R. Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. In *Proc. Of the 16th European Conference on Object-Oriented Programming*, pages 37-61, London, UK, 2002.

[24] C. Driver and S. Clarke. Distributed Systems Development: Can we Enhance Evolution by using AspectJ. In *Proc. Of the 9th International Conference on Object-Oriented Information Systems*, pages 368-382, Geneva, Switzerland, 2003.

[25] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Of the 4th International Conference on AOSD,* pages 123-134, Chicago, USA, 2005.