Aspect-Oriented Refactoring.

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

1. **Aspect-Oriented Refactoring.**
   - An Introduction to Refactoring.
   - An Example of AO Refactoring: Extract Method.
2. **Advanced Aspect-Oriented Refactoring Techniques.**
   - Extract Exception Handling.
   - Extract Concurrency Control.
   - Extract Interface Implementation
3. **Conclusions and References.**

Aspect-Oriented Refactoring
Refactoring and AOP ➜ Aspect-Oriented Refactoring

Refactoring is a process and a set of techniques to reorganize code while preserving the external behavior.

AOP allows of encapsulating crosscutting concerns in a system through use of a new unit of modularity (**aspects**).

Aspect-oriented refactoring synergistically combines these two techniques to refactor crosscutting elements.

Individually, refactoring and AOP

- share the high-level goal of creating systems that are easier to understand and maintain without requiring huge up-front design effort.

A combination of the two, aspect-oriented refactoring

- helps in reorganizing code corresponding to crosscutting concerns to further improve modularization and get rid of the usual problem symptoms: *code-tangling* and *code-scattering*.

Aspect-Oriented Refactoring
Refactoring and AOP ➜ Aspect-Oriented Refactoring (Cont'd).

Aspect-oriented refactoring goes beyond conventional refactoring techniques

- conventional refactoring modularize code into a clean OO implementation,
- the use of AOP squeezes out code that cannot be further refactored.

Aspect-oriented refactoring offers substantial improvement in a variety of situations.

Aspect-oriented refactoring offers many benefits:

- implementation of the same functionality with fewer lines of code;
- the code is easy to understand, highly consistent, and simple to change.

Aspect-Oriented Refactoring

Peculiarities of Aspect Oriented Refactoring.

Most of the AOP principles still apply to AO refactoring but often they acquire a particular relevance or a different perspectives.

Approach towards crosscutting functionality.

- In AOP, the focus is on adding a new crosscutting functionality on existing applications.
- In AO refactoring, we design and prototype a conventional solution, then we can design and implement aspects to encapsulate such functionality.

Applicability of aspect's crosscutting

- When implementing a crosscutting functionality using aspects, we restrict crosscutting to only selected parts of the system. Then we will increase the scope of the aspects.
- Refactoring aspects deliberately limit crosscutting to typically just a few methods or a class, and sometimes a package.

Aspect-Oriented Refactoring

Peculiarities of Aspect Oriented Refactoring (Cont'd).

Coupling consideration.

In AOP usage, it is desirable to minimize coupling between aspects and classes whereas in refactoring usage, the coupling issue receives less emphasis.
A refactoring aspect is a part of the target class' implementation and therefore it may use intimate knowledge from the class.

Placement of aspects.

In refactoring usage, since the aspects may have to change with the implementation, it is desirable to put aspects closer to the target module.

Aspect-Oriented Refactoring

An Example of AO Refactoring: Extract Method.

Aspect-oriented refactoring provides additional means to conventional refactoring techniques.

Consider the "Extract method" for refactoring

- it encapsulates logic into a separate method, while leaving calls to the method in potentially multiple places.

Instead, with the AO refactoring technique of "Extract method calls",

- you can take an additional step and refactor out even those calls into a separate aspect;
- you can also refactors out any exception handling code into a separate aspect.

There are several concrete occasions for such refactoring, e.g., class-level logging, security permission checks, persistence session management.

Aspect-Oriented Refactoring

An Example of AO Refactoring: Extract Method (Cont'd).

Symptoms.

The implementation has a duplicate piece of code in multiple places.

Classical Extract Method Refactoring.

- it encapsulates the duplicated logic in a new method; and
- it replaces each original piece of code with a call to the new method.

The problem is not completely solved yet, now the tangling code are the duplicated calls.

Aspect-Oriented Extract Method Refactoring.

It encapsulates those method calls in an aspect;

- a pointcut captures all the places the method should be called;
- advises that pointcut when to call the refactored method.

Aspect-Oriented Refactoring

An Example of AO Refactoring: Extract Method (Cont'd).

```
public class Account {
  private int _accountNumber;
  private float _balance;
  public Account(int accountNumber) { _accountNumber = accountNumber; }
  public int getAccountNumber() {
    AccessController.checkPermission(new BankingPermission("accountOperation"));
    return _accountNumber;
  }
  public void credit(float amount) {
    AccessController.checkPermission(new BankingPermission("accountOperation"));
    _balance = _balance + amount;
  }
  public void debit(float amount) throws InsufficientBalanceException {
    AccessController.checkPermission(new BankingPermission("accountOperation"));
    if (_balance < amount) throw new InsufficientBalanceException();
    else _balance = _balance - amount;
  }
  public float getBalance() {
    AccessController.checkPermission(new BankingPermission("accountOperation"));
    return _balance;
  }
  public String toString() {return "Account: " + _accountNumber;}
}
```


Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process.

We break this kind of refactoring process into two required steps followed by two optional steps.

At the end of each step, the behavior would match that of the original code, and therefore, you should run your unit tests to ensure that refactoring did not change any behavior.

Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process (Cont'd).

Step 1: Introduce a no-op refactoring aspect.

The purpose of this step is to create the required infrastructure (static and dynamic crosscutting), but add no crosscutting functionality.
This first step can be broken in three sub-steps:

- insert an empty aspect;
- define a pointcut to capture join points that need the refactored functionality;
- create no-op advice to the pointcut.

```
private static aspect PermissionCheckAspect {
  private pointcut permissionCheckedExecution() :
    (execution(public int Account.getAccountNumber()) ||
     execution(public void Account.credit(float)) ||
     execution(public void Account.debit(float) throws InsufficientBalanceException) ||
     execution(public float Account.getBalance())) && within(Account);
  before() : permissionCheckedExecution() {}
}
```


Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process (Cont'd).

Step 2: Introduce the crosscutting functionality.

We move code from the core class into the aspect.

- Introduce compile-time warnings (optional)

```
declare warning:
  call(void AccessController.checkPermission(java.security.Permission))
  && within(Account) && !within(PermissionCheckAspect) :
        "Do not call AccessController.checkPermission(..) from Account";
```

This will issue warnings in case a programmer, unaware of the aspect, introduces the refactored method back into the class.

- Add crosscutting functionality to advice.

```
before() : permissionCheckedExecution() {
  AccessController.checkPermission(new BankingPermission("accountOperation"));
}
```

- Remove method calls from the advised methods.

Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process (Cont'd).

```
public class Account {
  private int _accountNumber;
  private float _balance;

  public Account(int accountNumber) { _accountNumber = accountNumber; }
  public int getAccountNumber() {
    return _accountNumber;
  }
  public void credit(float amount) {
    _balance = _balance + amount;
  }
  public void debit(float amount) throws InsufficientBalanceException {
    if (_balance < amount) throw new InsufficientBalanceException();
    else _balance = _balance - amount;
  }
  public float getBalance() {
    return _balance;
  }
  public String toString() {return "Account: " + _accountNumber;}
}
```


Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process (Cont'd).

Step 3 (Optional): Simplify pointcut definition.

We redefine the pointcut to make it shorter and make it semantically more meaningful.

The pointcut defined in step 1 will:

- capture just the listed methods with the specified signatures;
- not capture method with the same functionality but added later or with a different signature.

This step is not mechanical and requires understanding of the interaction with the functionality and due diligence in implementation.

```
private pointcut permissionCheckedExecution() :
  (execution(public * Account.*(..)) &&
   !execution(String Account.toString())) && within(Account);
```


Aspect-Oriented Refactoring

An Example of AO Refactoring: the AO Refactoring Process (Cont'd).

Step 4 (Optional): Refactor the refactoring aspect.

We may refactor the refactoring aspect itself.

- Following the agile process wisdom, you will probably want to wait until you see some other class needing the same refactoring.

The process typically involves creating an abstract aspect and moving most of the functionality to it.

- The base aspect contains a few abstract pointcuts and methods.
- The concrete refactoring aspects for each refactored module extends this aspect and provides definition for pointcuts and implementation for the methods.

In a sense, such a refactoring is the AOP equivalent of "Extract superclass" refactoring, called "Extract base aspect".

Aspect-Oriented Refactoring

Extract Exception Handling.

Exception handling is a crosscutting concern that affects most nontrivial classes.

- Due to the structure of exception handling code (try/catch blocks), conventional refactoring cannot perform further extraction of common code.
- Each class (or a set of classes) may have its own way to handle exceptions encountered during execution of its logic.

With AO refactoring, you can extract exception handling code in a separate aspect.

There isn't any conventional refactoring technique to extract repeated try/catch blocks and the code associated with each.

Aspect-Oriented Refactoring

Extract Exception Handling (Cont'd).

```
package library;

public class LibraryDelegate {
  private LibrarySession _session;
  public LibraryDelegate() throws LibraryException { init(); }
  private void init() throws LibraryException {
     try {
        LibrarySessionHome home
           = (LibrarySessionHome)ServiceLocator.getInstance().
               getRemoteHome("Library", LibrarySessionHome.class);
        _session = home.create();
     } catch (ServiceLocatorException ex) { throw new LibraryException(ex); }
       catch (CreateException ex) { throw new LibraryException(ex); }
       catch (RemoteException ex) { throw new LibraryException(ex); }
  }
  public LibraryTO getLibraryDetails() throws LibraryException {
     try {
        return _session.getLibraryDetails();
     } catch (RemoteException ex) { throw new LibraryException(ex); }
  }
                              ...
```


Aspect-Oriented Refactoring

Extract Exception Handling (Cont'd).

```
                              ...

  public void setLibraryDetails(LibraryTO to) throws LibraryException {
     try {
        _session.setLibraryDetails(to);
     } catch (RemoteException ex) { throw new LibraryException(ex); }
  }

  public void addBook(BookTO book) throws LibraryException {
     try {
        _session.addBook(book);
     } catch (RemoteException ex) { throw new LibraryException(ex); }
  }

  public void removeBook(BookTO book) throws LibraryException {
     try {
        _session. removeBook(book);
     } catch (RemoteException ex) { throw new LibraryException(ex); }
  }
}
```


Aspect-Oriented Refactoring

Extract Exception Handling (Cont'd).

This aspect refactors out all the exception handling logic:

```
aspect LibraryExceptionHandling {
  declare soft : RemoteException
     : call(* *.*(..) throws RemoteException) && within(LibraryDelegate);
  declare soft : ServiceLocatorException
     : call(* *.*(..) throws ServiceLocatorException) && within(LibraryDelegate);
  declare soft : CreateException
     : call(* *.*(..) throws CreateException) && within(LibraryDelegate);

  after() throwing(SoftException ex) throws LibraryException
     : execution(* LibraryDelegate.*(..) throws LibraryException)
       && within(LibraryDelegate) {
     throw new LibraryException(ex.getWrappedThrowable());
  }
}
```

- Each **declare soft** statement causes any exception of the specified types thrown during a call matching the specified pointcut to be treated as a run-time exception;
- run-time exceptions do not require to be declared and trapped.

Aspect-Oriented Refactoring

Extract Exception Handling (Cont'd).

```
package library;
public class LibraryDelegate {
  private LibrarySession _session;
  public LibraryDelegate() throws LibraryException { init(); }
  private void init() throws LibraryException {
     LibrarySessionHome home
        = (LibrarySessionHome)ServiceLocator.getInstance().
           getRemoteHome("Library", LibrarySessionHome.class);
     _session = home.create();
  }
  public LibraryTO getLibraryDetails() throws LibraryException {
     return _session.getLibraryDetails();
  }
  public void setLibraryDetails(LibraryTO to) throws LibraryException {
     _session.setLibraryDetails(to);
  }
  public void addBook(BookTO book) throws LibraryException {
     _session.addBook(book);
  }
  public void removeBook(BookTO book) throws LibraryException {
     _session. removeBook(book);
  }
}
```

Aspect-Oriented Refactoring

Extract Concurrency Control.

Concurrency control is often a critically important, but equally hard to implement, crosscutting concern.

- a concurrency control implementation requires the code to be scattered over many methods;
- there are a few concurrency control patterns available, and although the concepts are reusable, their implementations aren't;
- AOP offers reusable implementations of those patterns, thus taking some pain out of concurrency control implementation.

In the next, we will consider a program that uses the read-write lock pattern.

- The concurrency pattern requires managing two kinds of lock: read lock and write lock;
- upon entering each method that only reads data, the read lock is acquired and that lock is released before leaving the method;
- the write lock is acquired and released in a similar manner for each method that modifies data.

Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

```
public class Account {
  private ReadWriteLock _lock = new ReentrantWriterPreferenceReadWriteLock();
  // ... constructors etc.

  public void credit(float amount) {
     try {
        _lock.writeLock().acquire();

        // ... business logic for credit operation ...

     } catch (InterruptedException ex) {throw new InterruptedRuntimeException(ex);
     } finally { _lock.writeLock().release(); }
  }

  public void debit(float amount) throws InsufficientBalanceException {
     try {
        _lock.writeLock().acquire();

        // ... business logic for debit operation ...

     } catch (InterruptedException ex) { throw new InterruptedRuntimeException(ex);
     } finally { _lock.writeLock().release(); }
  }
```


Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

```
  public float getBalance() {
     try {
        _lock.readLock().acquire();

        // ... business logic for getting the current balance ...

     } catch (InterruptedException ex) { throw new InterruptedRuntimeException(ex);
     } finally { _lock.readLock().release(); }
  }

  public void setBalance(float balance) {
     try {
        _lock.writeLock().acquire();

        // ... business logic for setting the current balance ...

     } catch (InterruptedException ex) { throw new InterruptedRuntimeException(ex);
     } finally { _lock.writeLock().release(); }
  }

  // ... more methods
```


Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

```
  public String toString() {
     try {
        _lock.readLock().acquire();

        // ... form the description string ...

     } catch (InterruptedException ex) { throw new InterruptedRuntimeException(ex);
     } finally { _lock.readLock().release(); }
  }
}
```

The solution uses a reusable `ReadWriteLockSynchronizationAspect` aspect

- it declares two abstract pointcuts: `readOperations()` and `writeOperations()`, and
- advices them to manage read and write lock.

Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

```
public abstract aspect ReadWriteLockSynchronizationAspect
    perthis(readOperations() || writeOperations()) {

  public abstract pointcut readOperations();
  public abstract pointcut writeOperations();

  private ReadWriteLock _lock = new ReentrantWriterPreferenceReadWriteLock();

  before() : readOperations() { _lock.readLock().acquire(); }
  after() : readOperations() { _lock.readLock().release(); }
  before() : writeOperations() { _lock.writeLock().acquire(); }
  after() : writeOperations() { _lock.writeLock().release(); }
  after() throwing(SoftException ex) throws InterruptedRuntimeException :
    readOperations() || writeOperations() {
      throw new InterruptedRuntimeException(ex);
  }
}

aspect SoftenInterruptedException {
  declare soft : InterruptedException : call(void Sync.acquire())
     && within(ReadWriteLockSynchronizationAspect);
}
```


Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

We refactor the concurrency control code for the `Account` class in a concrete sub-aspect of `ReadWriteLockSynchronizationAspect`, that

- provides a definition for the abstract pointcuts: `readOperations()` and `writeOperations()`;
- defines the read operations as all methods with their name starting in `get` as well as the `toString()` method;
- considers every other method as a write operation.

```
static aspect ConcurrencyControlAspect extends ReadWriteLockSynchronizationAspect {
  public pointcut readOperations()
     : (execution(* Account.get*(..)) || execution(* Account.toString(..)))
       && within(Account);

  public pointcut writeOperations()
     : (execution(* Account.*(..)) && !readOperations())
       && within(Account);
}
```


Aspect-Oriented Refactoring

Extract Concurrency Control (Cont'd).

```
public abstract class Account {

  // ... no _lock variable here (compared to not refactored listing)
  // ... constructors etc.

  public void credit(float amount) {
     // ... business logic for credit operation ...
  }

  public void debit(float amount) throws InsufficientBalanceException {
     // ... business logic for debit operation ...
  }

  public float getBalance() {
     // ... business logic for setting the current balance ...
  }

  public void setBalance(float balance) {
     // ... business logic for getting the current balance ...
  }

}
```


Aspect-Oriented Refactoring

Extract Interface Implementation.

The refactoring technique of "extract interface" allows improved decoupling of clients from implementations.

- if more than one class implements that interface, you may end up duplicating code required to implement the interface.

With AO refactoring, you can take the idea further and avoid any duplication.

Let's consider the `ServiceCenter` interface

```
public interface ServiceCenter {
  public String getId();
  public void setId(String id);
  public String getAddress();
  public void setAddress(String address);
}
```

Aspect-Oriented Refactoring

Extract Interface Implementation (Cont'd).

The ATM class implements the ServiceCenter interface.

```
public class ATM extends Teller implements ServiceCenter {
  private String _id;
  private String _address;

  public String getId() { return _id; }
  public void setId(String id) { _id = id; }
  public String getAddress() { return _address; }
  public void setAddress(String address) { _address = address; }
                        ...
}
```

Other classes, as well as the ATM class, could have similar implementations of the ServiceCenter interface.

- Each class repeating the code to implement the ServiceCenter interface.
- Duplicated code could be avoid by creating the default implementation of the interface and make the classes extend the default implementation, but we do not have multiple inheritance.

Aspect-Oriented Refactoring

Extract Interface Implementation (Cont'd).

With AO refactoring, we introduce the default implementation into the ServiceCenter interface using a nested aspect.

```
static abstract aspect implementation {
  private String ServiceCenter._id;
  private String ServiceCenter._address;

  public String ServiceCenter.getId() { return _id; }
  public void ServiceCenter.setId(String id) { _id = id; }
  public String ServiceCenter.getAddress() { return _address; }
  public void ServiceCenter.setAddress(String address) { _address = address; }
}
```

The aspect introduces data members as well as methods with the default implementation of the ServiceCenter interface.

- By nesting this aspect into the interface, any class that implements the interface automatically inherits the default implementation.

Aspect-Oriented Refactoring

Extract Interface Implementation (Cont'd).

Benefits.

- The implementing classes no longer include boilerplate code to implement the interfaces.
- The implementing classes can still override the default implementation provided by the aspects.

Variations.

The implementing classes could choose whether to inherit the default implementation provided by the aspect or not.

- By creating, say, ServiceCenterDefaultAspectImpl, a sub-interface of the original interface and let the aspect introduce the default implementation to this interface.

The implementing classes will inherit the default implementation only if they declare themselves to implement such a sub-interface.

Aspect-Oriented Refactoring

Conclusions.

Many other refactoring techniques can benefit of aspect-orientation, few examples are:

Replace override with advice.

It is often required to augment additional common behavior to many methods of a class.

- A typical solution is to create a subclass and override methods to perform some additional logic.
- AO solution, you can use an aspect to advise the needed method with additional logic.

Extract Lazy initialization.

Lazy initialization of expensive resources is a common optimization technique.

- It requires checks for uninitialized resources in every place that uses those resources and causes code-scattering and -tangling.
- The use of getter methods helps but do not prevent from the direct access to the resource.
- With AO refactoring, you can advise read access to the resource (using a get() pointcut) to initialize it.

References

- Ramnivas Laddad.
  Aspect-Oriented Refactoring Series: Overview and Process.
  TheServerSide.com, December 2003.
  Available at http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1.
- Ramnivas Laddad.
  Aspect-Oriented Refactoring Series: The Techniques of the Trade.
  TheServerSide.com, December 2003.
  Available at http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart3.

Aspect-Oriented Refactoring

Utilizing Static Crosscutting to Flag Mismatched Join Points.

We create a temporary pointcut that captures a larger sets of join points according to their semantics.

- We could capture all public methods of the `Account` class.

```
pointcut tmp() : execution(public * Account.*(..)) && within(Account);
```

Now, we will use a construct that produces errors if two pointcuts do not match exactly the same set of join points.

- This is feasible when both pointcuts are statically determinable.

```
declare error:
  (!permissionCheckedExecution() && tmp()) ||
  (permissionCheckedExecution() && !tmp()) : "Mismatch in join points captured";
```

Any errors issued by compiler imply that the two pointcuts aren't equivalent.

- We will get an error since `tmp()` captures the `toString()` method but `permissionCheckedExecution()` does not.

Aspect-Oriented Refactoring

Utilizing around() to Remove the Factored Calls.

In a larger system, you might want to write an around advice to make the factored method calls a no-op and test the result before actually removing the code.

- For example, you can introduce the following advice to nullify calls to the `AccessController.checkPermission()` method from the `Account` class.

```
void around() :
  call(void AccessController.checkPermission(java.security.Permission)) &&
     within(Account) && !within(PermissionCheckAspect) {
          // do NOT call proceed()
  }
```

Once you successfully complete the testing, you may remove the advice.