Patterns of Functional Programming for Simpler Design

Arnaud Bailly - Cédric Pineau

2012-11

This page intentionally left blank

Contents

Introduction

This little booklet aims at summarizing, with some level of details, common *Functional Programming Patterns* and how they apply to Object-Oriented programming. These patterns were selected from the existing literature, both in OO programming and FP programming, with an aim to be directly applicable to day-to-day coding problems. Due to the limitation of the author, they also were selected on the basis of their simplicity and commonality in both paradigms, eg. some of them are quite obvious and part of the OO folklore like (Function object) while some other are nearly unknown in OO programming but quite common in FP (Monad).

For each pattern, I have tried to provide meaningful and relevant examples, when possible from real-world codebases that effectively use these patterns. Some more advanced patterns have been left over for another edition of this booklet. The main sources used are:

- Domain Driven Design[1], E.Evans, Addison-Wesley, 2004

[1] http://www.domaindrivendesign.org/

- Domain Specific Languages[2], Martin Fowler, Addison-Wesley 2008
- A Functional Pattern-System for Object-Oriented Programming[3], Thomas Kühne, PhD Thesis, 1998
- Haskell Wiki[4]

The pattern format used here is straightforward:

- *Intent*: Why would anyone want to use this pattern
- *Motivation*: Which problem are we trying to solve
- *Implementation*: How this pattern solves the given problem
- *Tradeoffs*: Pros and cons of this pattern
- *Related Patterns*: How this pattern relates to others
- *References*: Where this pattern comes from

The Patterns

Expression Builder

- *Intent*: An object or family of objects that provides a fluent interface over a normal command-query API
- *Related Patterns*:
  - Cumulative Factory[5]: Use when one needs to provide *setters* semantics to some immutable objects
  - Builder Pattern: Classical creational pattern from GoF, provide a mutable layer to simplify construction of complex objects
  - Zipper: Provide an alternative way of constructing complex immutable values, filling it from an intially "empty" structure
  - Fluent Interface[6]
  - Closure of Operations: When working on a single object type with immutable value objects then Fluent interface is a Closure of Operation
- *Motivation*:

Quoting Martin Fowler's *Domain Specific Languages* book:

[2] http://martinfowler.com/books/dsl.html
[3] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1134&rep=rep1&type=pdf
[4] http://www.haskell.org/haskellwiki/Haskell
[5] http://wiki.apidesign.org/wiki/APIDesignPatterns:CumulativeFactory
[6] http://www.martinfowler.com/bliki/FluentInterface.html

> APIs are usually designed to provide a set of self-standing methods on objects. Ideally, these methods can be understood individually. I call this style of API a command-query API [..] An Expression Builder provides a fluent interface as a separate layer on top of a regular API. This way you have both styles of interface and the fluent interface is clearly isolated, making it easier to follow.

Expression Builder (or other forms of Builder patterns) should be the default interface to creation of complex data structures like expressions and syntactic trees and other complex Value objects.

- *Implementation*:

An implementation of Expression Builder for constructing Value objects should provide various forms of *copy constructors* as public API methods or *friend* Builder implementation like in the following example:

```
public final class User {

   public static class Builder {
       private String name;
       private Address address;

       public Builder name(String name) {
          this.name = name; return this;
       }

       public Address address(Address address) {
          this.address = address; return this;
       }

       public User build() {
          return new User(name, address);
       }
   }

   private User (String name, Address address) {
      this.name = name; this.address = addres;
   }

   ...
}
```

- *References*:
  - Domain Specific Languages[7], Martin Fowler, Addison-Wesley 2008, pp.343-355

[7] http://martinfowler.com/books/dsl.html

Immutable Value Object

- *Intent*: Represent "things" with properties and no identity as immutable objects
- *Motivation*:

When working with complex domain models, we need to distinguish between *entities*, that is objects which have a well-defined *identity* within the system that needs to be tracked, and *values* which describes as an arbitrarily complex structure undistinguishable characteristics of the entities. To put it in the words of Eric Evans:

> Value Objects are instantiated to represent elements of the design that we care about only for what they are, not for who or which they are

Value objects are by essence *immutable*, otherwise this would contradic their absence of identity. A value object should be *equal* to any other object instance with same values irrespective of where it has been built and which memory location it references.

- *Implementation*:

Value objects in Object-oriented programming are "easily" implemented as immutable objects and classes. In java using Guava:

```
@Immutable
public class User {
  private final String email;
  private final String firstName;

  public User(String email, String firstName) {
    this.email = email;
    this.firstName = firstName;
  }

  public String getEmail() {
    return email;
  }
}
```

Note that immutability can be rather tricky to implement in languages such as Java as quite a number of data structures and common objects are actually mutable. Moreover, using some "dirty tricks" involving bytecode engineering it is possible to make objects mutable: No support for immutability is provided by the language or the underlying runtime.

- *References*:
  - Whole Value[8]
  - Domain Driven Design[9], E.Evans, Addison-Wesley, 2004, p.97-103

[8] http://c2.com/ppr/checks.html#1
[9] http://www.domaindrivendesign.org/

Function Object

- *Intent*: Encapsulate a function with an object. This is useful for parameterization of algorithms, partial parameterization of functions, delayed calculation, lifting methods to first-class citizens, and for separating functions from data.
- *AKA*: Blocks, closures, Functionals,...
- *Motivation*:

Behaviour parametrization is a cornerstone of almost all software of significant size: One recurringly needs a way to change the behaviour of objects within the system depending on some conditions. Collections are a very common use case, for example when *filtering* elements of a collection:

```
List<User> users = ...
// filter all ”Johns” without Function object
List<User> johns = Lists.newArrayList();
for(User user : users) {
  if(user.firstName().equals(”John”)) johns.add(user);
}

// filter with Function Object in Guava
Predicate<User> firstNameIsJohn = ...
Iterable<User> johns = Iterables.filter(users, firstNameIsJohn);
```

Other uses within collections involve *sorting* with a specific ordering relation, transforming elements with a *map* and many more. The use of Function Object prevents duplication, encapsulation breach by exposing the internals of iteration logic, awkward reuse.

Beyond collections, Function Object has countless potential usages: Reifying business rules within forms or applicative logic, decoupling, customizing and reusing parts of an algorithm, ...

- *Implementation*:

Where closures or blocks are not directly supported by the language (eg. Java), typical implementation involves:

- The definition of a *Function* or equivalent interface offering a single method *apply()*,
- Possible definitions of *Function2*, *Function3* and the like for different arities,
- Inner or anonymous subclasses for providing concrete implementation.

Function is typically parametric in its input and output type to remove cruft of casting and opaque Object handles. Note that Java does not permit variance annotation which would be handy in this case as a Function is usually contravariant in its arguments and covariant in its return type. Definition of Function1 in scala is:

```
trait Function[-I,+O] { def apply(I : input) : O }
```

- *Advantages*:
  - Provide *closures* as first-class objects thus allowing encapsulating behaviours and moving them around
  - Decouple behaviour from data and promotes *black-box* reuse: Neither it nor its client need to know implementation details of one another
  - Promotes reuse through *composition* of functions instead of inheritance or delegation, which is much more flexible and done at runtime (possibly with compile-time type-checking)
- *Drawbacks*:
  - When language has poor support for closures, it yields to awkward constructions that may clutter the code.
  - Multiplying anonymous function objects can yield to code which is difficult to debug: Stack traces get ladden with lots of `apply(xxx)` calls which do not relate obviously to some meaningful piece of code
  - Some performance might be lost by adding more calls, deepening stack's depth and maintaining more memory for closures.

Consider the following definitions of the same predicate in javascript and scala:

```
var firstNameIsJohn = function(user) { return user.firstName === ”John”; }
```

```
val firstNameIsJohn = (u: User) => u.firstName == ”John”
```

- *Related Patterns*:
  - Command[10]: Parameterizes the actions of some object by delegating it to a reified *method object*. Commands are mostly procedural in natures, Function Object is a more general approach
  - Strategy[11]:
  - Functor: Assumes the existence of Function Object or some other form of closure to *lift* it into some context
- *References*:
  - A Functional Pattern-System for Object-Oriented Programming[12], Thomas Kühne, PhD Thesis, 1998

[10] http://www.vincehuston.org/dp/command.html
[11] http://www.vincehuston.org/dp/strategy.html
[12] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1134&rep=rep1&type=pdf

Lazy Object

- *Intent*: Defer computation to the latest possible time by encapsulating it in an object that can be computed at any time
- *AKA*: Suspension, Thunk, Stream, Pipes & Filters, Pipeline
- *Motivation*:

Imperative programming usually enforces a specific evaluation sequence to evaluate some piece of code. In Java, method call semantics is *call-by-value* which implies all arguments to a method need to be evaluated before the method itself may be called. But call-by-name, which only passes around references of values also needs to evaluate its arguments. The call-by-need semantics as implemented in Haskell promotes so-called *lazy evaluation*: A function evaluates its arguments only when it needs them. When evaluating a function containing an `if-then-else` construct for example, one might not need some argument to the function which is only used in one branch of the alternative.

Laziness is also extremely useful when working with potentially infinite data structures, like iterating over a *stream* of data. Strict adherence to call-by-value would entail a non-terminating program before any computation could be ever done. For the same reason, working with costly operations or sequence of operations should be done lazily. Finally, Lazy Object makes it possible to implement data-flow programming in languages not supporting this paradigm directly: Computation is carried in steps involving the evaluation of values if and only if they are needed, thus proceeding along the *flow of data dependencies*.

- *Implementation*:

When implementing Lazy Object, one can also have the added benefit of *memoization*, eg. caching the result of the (lazy) computation for further use.

```
public abstract class Lazy<A> {
  private A cached;

  public abstract A value() {
     if(cached == null) {
         cached = compute();
     }
     return cached;
  }

  protected abstract A compute();
}
```

Streams provide a prototypical example where Lazy Object shines. Its interface (like lists) provide access to its head, a value of some type, and another stream, its tail. The stream itself is a lazy object that does not compute its tail until needed.

```
interface Stream<A> extends Iterable<A> {
  A head();
  Stream<A> tail();
  boolean isDone();
}
```

A stream may be *done* or not, in which case it holds a value. Its *iterator* (not shown) simply loops over the subsequent streams as requested by the client, until the stream is done.

```
public abstract class LazyStream<A> implements Stream<A> {
   private final A a;
   public LazyStream(A a) { this.a = a;}
   public A head() {
     return a;
   }

   public Stream<A> tail() {
     return next();
   }

   public boolean isDone() { return false; }

   public Iterator<A> iterator() { ... };
   protected abstract Stream<A> next();
}
```

An empty stream has no value and tails on itself:

```
public final class EmptyStream<A> implements Stream<A> {
  public A head() { throw new IllegalStateException("stream is empty"); }

  public Stream<A> tail() { return this; }

  public boolean isDone() { return true; }
}
```

- *Related Patterns*:
  - Immutable Value Object: Memoization is most meaningful when computed values are immutable hence have no identity
- *References*:
  - A Functional Pattern-System for Object-Oriented Programming[13], Thomas Kühne, PhD Thesis, 1998
  - Haskell Language Report, 2010

[13] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1134&rep=rep1&type=pdf

Monoid

- *Intent*: Group together a way of composing values within some type or data structure
- *Relations*:
  - *Closure of Operations*: Use when operations may not respect monoidal laws
  - *Transfold*: Use in conjunction with a Monoid when values permit it to abstract away the details of what we are folding over
  - *Monad*: A particular form of "Monoid with a Context" or "Monoid of Functors"
  - *Lazy Object*: Lazily applying the monoid's operations can provide more efficient implementations
- *Motivation*:

A client might want to be able to aggregate different kind of values without taking care of their implementation details, typically when using numbers or implementing numeric applications. For example, a client wants to compute the average of some number series where number can either be integers, reals or even complex values. Provided there exist some implementation of integral division (`div(int)`) for each type, considering each type as monoid simplifies client code:

```
Monoid<?> monoid = Monoid.monoid(sum, zero);
...
List<? extends Numeric> numbers = retrieveNumbers();
...
monoid.sumRight(numbers).div(numbers.size());
```

Technically, a Monoid is a set with a `sum()` operation and a distinguished neutral element `e` respecting associativity and neutral operation with `e`. Non numeric values can give rise easily to monoids provided they respect the basic laws. Typical examples are:

- Strings with operator + and empty string
- Iterables with `concat` and empty Iterable
- Matrices with operator * and unit matrix
- ...
- *Advantages*:
  - Abstracts away the detail of composing values of different types when all one is interested in is the aggregation
  - Decouple composition from their implementation: A single type can provide different implementations of monoid representing different ways of composing values
- *Drawbacks*:
  - ??

Option

- *Intent*: Signals the possibility of a non-existing value (eg. Null object)
- *Relations*:
  - *Void Value*: Can be used instead of *Option* where all that is needed is representing some unique initial/terminal object
  - *Functor*: An Option usually implements Functor pattern to be able to transparently transform contained values
- *Motivation*:

When passing around references to objects, it might be tempting to use `null` to represent the absence of value:

```
public Credentials login(String login, String password) {
  if(!passwordDb.check(login, password) {
     return null;
  }
  ...
}
```

This then entails the need to check the possibility of a reference to be null all over, cluttering the code with statements like one:

```
public void login(User user, Credentials credentials) {
  if(credentials == null) {
     throw new NotLoggedInException(...);
  }
  ...
}
```

An **Option** wraps a type and exposes its content:

- Either as a generic *None* value, denoting an empty option,
- Or as containing a single value *Some(value)*.

In Guava this type is named **Optional** and can be constructed easily, including to wrap null values (eg. returned from some legacy library call). It provides various methods such as defaulting to a value when empty or transforming its content when non-empty using some function (since 12.0).

```
Optional<String> optional = Optional.fromNullable(aString);
...
return optional.or(defaultValue);
```

- *Advantages*:
  - Removes the need for null, make the code more concise and less error-prone to the dreaded NullPointerException (or whatever it is called in your language)
  - Thin layer over the wrapped object
- *Drawbacks*:
  - Added storage overhead: Adds one more indirection and object for each held reference
  - Abstraction leak: Using an Option quickly "contaminates" the codebase for data types that are very common, thus leading to cumbersome code. It can imposes some dependencies on clients of the code

Lens

- *Intent*: Decouple the operations on element contained in an object from its concrete structure
- *Motivation*:

When Objects are immutable, traversing them when we want to modify/update a value is cumbersome

Example: Update ZIP code of a User

```
User user = ...
Address address = user.address;

User updated = user.builder().(
   address.builder().zip("12345").build()
  ).build();
```

We would like to have a way to express this update as a composable transformation over Users and addresses.

Lenses provide this feature as a pair of functions: one for *get*, one for *set*.

```
User user = ...

Lens<User> userZip = lens("zip", Address.class).in(lens("address", User.class));

User updated = userZip.set(user, "12345");
```

- *Advantages*:
  - Change logic is manipulable as a first class object, thus decoupling business logic and algorithms from the concrete structure of objecs. One only has to manipulate lenses, compose them, then feed objects at a later stage

  - Depending on implementation, it can be typesafe, checked at compile-time and/or generated automatically from an Object's structure
  - Provides copy-on-write semantics when changing immutable objects

- *Drawbacks*:

  - Awkward to implement in Java, need use of reflection or compile-time magic or bytecode instrumentation to work efficiently

- *References*:

Zipper

- *Intent*: Provide a way to traverse and modify immutable structures
- *AKA*: Derivatives, One-hole Structure
- *Relations*:

  - *Lens* pattern: Both let user express and compose changes on an object. But the zipper keeps track of a context thus allowing more complex operations

- *Motivation*:

While modifying complex or nested immutable data structures, one needs to keep track of the context of the traversal and operations on the data to be able to reconstruct a new data which is equivalent to the old one but for some changes. This bookkeeping is tedious, cumbersome and error-prone.

Example: update all postings related to a user using some conversion function

```
User user = ...
Function updatePostings = ...

User newUser = UserZipper.zip(user).postings().update(updatePostings).build();
```

A Zipper encapsulates some navigation's state while providing a way to modify the underlying value. It is usually made of:

- A current location or *focus*
- Context(s) representing the rest of the data structure:

  - For lists, might be the left and right segments around the focus,
  - For binary trees might be the left, right and parent trees around the focus.

A Zipper is tailored to a specific base type and provides the following common operations:

- Initialization: Starts the Zipper at the "beginnning" of the wrapped value.
- Moving around: Returns a new Zipper with changed focus depending on the navigation property of the underlying structure (eg. `left()/right()` for lists). Note that left/right can be enough for all structures if they support some form of linearization
- Modifying focused value: Uses an update function to create a new zipper with value under focus updated
- Extraction: Extracts a complete (possibly changed) value from the Zipper
- *Advantages*:
  - Remove bookkeeping burden from the developer while retaining the ability to point to some part of structure
  - Simplify updates of immutable structure
  - Decouple transformations of values within structures from the structure holding them
- *Drawbacks*:
  - Each type need a specific Zipper construct as its navigation is specific
- *References*:
  - Haskell Wiki
  - Original Zipper article by G.Huet
  - Functional Java 3.0 library

Functor

- *Intent*: Provide a generic interface for containers of singly typed values decoupling client code from container's implementation details
- *AKA*: Generic Container
- *Related Patterns*:
  - Function Object: Provides the foundations for Functors
- *Motivation*:

When one works with a collection of objects and wants to do some bulk transformation or operation on all the objects within the collection, the structure of the collection is irrelevant, as well as its ordering or any other characteristics that distinguishes this collection from any other type of collection: It acts as a *container* for some value or values. It is enough then to be able to delegate applying the transformation to the container itself without any knowledge of how it is actually implemented.

More generally, any structure or object that encapsulates another (type of) object, providing some additional properties and methods, may as well provide the capability to apply some transformation on the

object it contains without revealing its internal structure or imposing any specific requirement on its client. For example, if one can construct a `Future<String>` instance, and wants to convert the returned `String` value, if it exists, to an `Integer`, it would be too stringent to require the client to *explicitly wait* on the Future's outcome. One would like to be able to *apply* the *parseInteger* function to the `Future<String>` yielding a `Future<Integer>` which will provide the transformed value, once or if available.

In other words, a Functor `F` (a generic type) provides the ability, for any instance `F<A>`, to apply a function `f : A → B` on the container, yielding an instance of `F<B>`.

- *Implementation*:

Functor implementations are specific to the concrete type that encapsulates one or more value of the same type. It should adhere to the following interface:

```
public interface Functor<A> {
  <B> Functor<B> map(Function<A,B> f);
}
```

Note the `map` method can also be viewed as transforming a function `f: A → B` into a function `F<f> : F<A> → F<B>`, which is the traditional functional language presentation. This method should respect composition law of *functions*: Applying the composition of two functions or composing the result of two maps should yield the same final value.

Here is an example implementation for Futures, provided as a static `map` method. Note this implementation requires an `ExecutorService` to preserve asynchronous semantics of Future's execution:

```
class Functors {

  public static <A,B> Function<...> map (final Function<A,B> f,
                                         final ExecutorService executor) {
     return new Function<Future<A>, Future<B>>() {
        public Future<B> apply(final Future<A> future) {
          return executor.submit(new Callable<B>() {
               public B call() {
                 A a = future.get();
                 return f.apply(A);
               }
            }
        }
     };
  }
```

This allows a client to write code that is agnostic with respect to the execution policy of the values it manipulates provided they do not try to access directly the value (if they use the Functor interface):

```
// select only the address part of a User object
public Functor<Address>  extractAddresses(Functor<User> users) {
  return users.map(new Function<User,Address>() {
    public Address apply(User user) {
      return user.address;
    }
  };
}
```

- *Advantages*:
  - Promotes better encapsulation by allowing clients to ignore implementation details of the structures they use
  - Allows reusing transformations (eg. Functions) in various and even unforeseen contexts
- *Drawbacks*:
  - Adds another layer of indirection and possibly complexify the execution path of the code
- *References*:
  - Guava's `transform` method for `Iterables`, `Optional`, `ListenableFuture` and other types
  - Functional Java

Applicative

- *Intent*: Allows working with multiple arity functions in an enclosing transparent context
- *Related Patterns*:
  - *Functor*: Applicative is a "generalization" of Functor to multiple arity functions
  - *Monad*: Applicatives imposes less structure than monads, at the expense of less guarantees (eg. no sequencing)
  - *Lazy Object*: An Applicative defers computation by allowing partial application on a Function object
- *Motivation*:

Initial motivation for implementing the applicative pattern within the functional programming community came with *parsing combinators*. Parsing combinators libraries makes it possible to describe (LL) grammars directly within the host language through the use of various *combinators*, eg. closed functions that operate on some state and consumes part of an input stream. In this context, one often needs to apply some function with more than one arguments on two fragments resulting from the parsing, *while preserving* parser's state.

Here is an example syntax rule describing a function call written using jParsec[14] library, where the `FunctionCall` constructor needing 2 arguments is applied to the result of parsing a `VARIABLE` and an `EXPRESSION` (parens are ignored after parsing):

[14] http://jparsec.codehaus.org

```
private static final Parser<FunctionCall> METHOD_CALL =
        curry(FunctionCall.class).sequence(VARIABLE, //
            skip(OPEN_PAREN, "open ("), //
            EXPRESSION.lazy().sepBy(COMMA), //
            skip(CLOSE_PAREN, "close )"));
```

Note that the type of `METHOD_CALL` is a `Parser<FunctionCall>`, that is a *Parser* object that recognizes a *FunctionCall* object or fails. This parser can be compoesed with other parsers to produce an arbitrarily complex grammar.

Another example is the *Stream* container: When considered as an Applicative, it can be used to *zip* together a stream of functions and a stream of values.

More generally, an Applicative is useful when one wants to be able to apply Function objects that are encapsulated within a Functor (an Applicative is always a Functor). In other words, an Applicative App provides a way to applly a `App<Function<A,B>>` to an `App<A>` yielding an `App<B>` object, or equivalently transforming the nested function into a `Function<App<A>,App<B>>` function.

- *Implementation*:

Here is the example implementation for *Streams*, similar to the one from Functional Java:

```
  public final <B> Stream<B> apply(final Stream<Function<A, B>> sf) {
    if(!sf.isDone()) {
      return new LazyStream(sf.head().apply(this.a)) {
         public Stream<B> next() {
           return tail().apply(sf.tail());
         }
      };
    } else {
      return new EmptyStream();
  }
}
```

- *Advantages*:
  - Provides yet another decoupling between the structure (or *shape*) of a container of objects and operations on these objects
- *Drawbacks*:
  - Abstracts away too much from the concrete business operation details
- *References*:
  - Applicative Programming with Effects[15], Conor McBride and Ross Paterson
  - Essence of the Iterator Pattern[16]: A scala implementation of original paper in Haskell

[15] http://www.soi.city.ac.uk/~ross/papers/Applicative.html
[16] http://etorreborre.blogspot.fr/2011/06/essence-of-iterator-pattern.html

Monad

- *Intent*: Compose operations while preserving transparently an enclosing context
- *AKA*: Triple, Kleisli triple
- *Related patterns*:
  - Applicative: Proper monads are normally also Applicatives, hence support the same kind of operations
  - Function Object: Support abstracting over functions
  - Monoid: A Monad may be viewed as a kind of monoid on encapsulated values
  - Fluent Interface: A monad can be used to provide some form of Domain Specific Language
  - Lazy Object: Monadic code usually works in two phase, first constructing some sequencing of operation then executing it.
- *Motivation*:

Monads originated in theoretical works on *pure functional* programming languages as a way to formalize side-effects in languages that are not supposed to support them. They received lot of attention from this community, especially in Haskell programming circles as they permitted to express various forms of "imperative" (e.g. enforcing execution order) programming. In particular, complex loops supporting filtering and nesting, called *for-comprehensions* are based on constructions provided by monads.

For example, here is in Scala a way to compute all pairs of elements of a list whose sum is an even number:

```
scala> def sumIsEven(xs : List[Int], ys : List[Int]) : List[(Int,Int)] =
     |     for ( x <- xs;
     |           y <- ys;
     |           if (x + y) % 2 == 0) yield (x,y)
sumIsEven: (xs: List[Int], ys: List[Int])List[(Int, Int)]

scala> sumIsEven(List(1,2,3,4), List(4,5,6,7))
res1: List[(Int, Int)] = List((1,5), (1,7), (2,4), (2,6), (3,5), (3,7), (4,4), (4,6))
```

Here the monad involved is the *list monad* which represents computations with multiple outcomes. More generally, monads are all kind of structures, containers or contexts that support some form of *shape-preserving* composition: Options, Futures, Lists, Functions with argument type fixed,...

Another motivating example might be found when working with some kind of external resource requiring specific form of management, like database operations and connections. Typically, database access is done in transactional units, where a sequence of queries and commands is delimited by a transaction that provides atomicity (among other properties). Neal Ford[17] proposes to use *Composed Method* pattern and *Single Level Abstraction Principle* to code databased operations within JDBC encapsulating the transactional context. Things get interesting when one wants to *compose* operations, and the proposed method falls short

[17] http://www.ibm.com/developerworks/java/library/j-eaed4/index.html

in this respect: One cannot compose 2 transactional operations and get a single transactional operations, effectively merging the transactional contexts.

Monads pattern specifically address this point as they can be arbitrarily composed while preserving the transactional context: The transaction occurs only when the monad is *run*. Here we can compose queries that are totally unrelated but operate on the same *monad* and thus are composable. The final `commit` call is where the transaction is discharged and the queries effectively executed within a single transactional context.

```
public void placeOrder(DatabaseConnection databaseConnection,
          final ShoppingCart cart,
          final String userName) throws SQLException {
  final OrderOperations order = new OrderOperations();
  final InventoryOperations inventory = new InventoryOperations();
  unit(cart).bind(order.createOrder(userName))
    .bind(order.addOrderFrom(userName))
    .bind(inventory.prepareDispatching(userName))
    .commit(databaseConnection);
}
```

- *Implementation*:

A Monad is specified by two operations, `unit` (aka. `return`) which allows injecting arbitrary values within the Monad, and the `bind` which sequences an *Function* within the Monad, propagating the context. Here is the detailed implementation of a very simple transactional monad for database operations.

The `DatabaseOperations<A>` is our Monad of interested: It is based on a `DatabaseConnection` that provides all the machinery to access the database using JDBC (it could equally have been injected as an instance variable). The method `run` triggers execution of the

The *unit* is represented by a subclass which simply encapsulates a value as a form

```
public abstract class DatabaseOperations<A> extends DatabaseConnection {

  private static class Return<A> extends DatabaseOperations<A> {

    private final A a;
    ...
    @Override
    protected A run() throws SQLException {
      return a;
    }

  }
```

Implementation of the `bind` operation is more interesting: It takes as input an existing monadic value (eg. an instance of `DatabaseOperations<A>`) and Function object (here denoted `Command`) taking as input an `A` and returning another monadic value, and when run it applies the *command* on the result of running the first monadic value, then run the resulting Monad.

```
  private static class Bind<A,B> extends DatabaseOperations<B> {

    private final DatabaseOperations<A> a;
    private final Command<A,DatabaseOperations<B>> command;

    public Bind(DatabaseOperations<A> a, Command<A,DatabaseOperations<B>> command) {
      this.a = a;
      this.command = command;
    }

    protected B run() throws SQLException {
      DatabaseOperations<B> txA = command.execute(a.run());
      return txA.run();
    }

  }
```

Finally, we get to the `commit` operation which starts the ball rolling from an initial Monad: The monad is *run* within a transactional context that is eventually cleaned up. This effectively hides all the necessary plumbing from the end-user who simply has to take care of properly sequencing and running its queries. Moreover, this promotes reuse as fragments of SQL "scripts" can be developed independently then later on composed to produce some larger operations.

```
  public final A commit(DatabaseConnection db) throws SQLException {
    db.setupDataInfrastructure();
    try {
      A a = run();
      db.completeTransaction();
      return a;
    } catch (SQLException sqlx) {
      db.rollbackTransaction();
      throw sqlx;
    } finally {
      db.cleanUp();
    }
  }
}
```

- *Advantages*:
  - Makes complex contextual operations easily composable and reusable
  - Isolate client code from the details of managing the context of the computation thus focusing on implementing business logic
  - Underlying logic can change without impacting client code. It is even possible to *stack* Monads on top of one another, as a form of *Strategy* pattern, to change the sequence's behaviour without changing its "shape"

  - Allow quick and safe development of small Domain Specific Languages, leveraging other patterns like Lazy object, Fluent Interface or Method Chaining.

- *Drawbacks*:

  - Needs a somewhat complex machinery to work properly (implementations of bind and unit, meaningful composition, context managment) without language support. This can clutter the codebase and appears overengineered for small applications
  - Hiding too much of the context might lead to unexpected and surprising behaviour for the client code level

- *References*:

  - There are really countless *monad tutorials* to be found on the web, and quite a number of papers explaining their origin, properties and use
  - Erik Meijer about LINQ[18] provides a detailed account of usage of Monad pattern in the context of the LINQ infrastructure and C# language

Other Patterns

From *Domain-Driven Design*

- *Side-effect Free Functions*: This is self-explanatory! When dealing with a complex Domain and a web of objects, side-effects free functions favor composition, reuse, and ease reasoning by limiting complexity to what is done *locally* (pp 250-254)

- *Closure of Operations*: Provide methods on objects returning same or other object transformed. Ä closed operation provides a high-level interface without introducing any dependency on other concepts (pp.268-270)

From *A Functional Pattern System*

- *Transfold*: Process the elements of one or more aggregate objects without exposing their representation and writing explicit loops. Allows expressing some kind of map-reduce[19] using higher-order functions like *fold* and *map*

- *Void Value*: Raises `null` to a first-class value. Among many other benefits, this allows to treat void and non-void data uniformly. Even Sir Tony Hoare himself acknowledges[20] using null was a mistake!

- *Translator*: Add semantics to structure with heterogeneous elements without changing the elements. This is a replacement for the classical *Visitor* pattern, using reflection or other form of *Runtime Type Information* to replace double-dispatching.

[18] http://cacm.acm.org/magazines/2011/10/131398-the-world-according-to-linq/fulltext
[19] http://code.google.com/edu/parallel/mapreduce-tutorial.html
[20] http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

From Functional Programming Folklore

- *Iteratee*: A different form of producer-consumer pattern where the consumer (the *Iteratee*) controls how it is fed input by the producer (the *Enumerator*). Provides efficient and elegant ways to process streams of data and compose those processors
- *Co-monad*: Provide a way to generate values within a context. A close cousin to the Zipper and, obviously, to the Monad.