



# Refactoring mit monadischen Transaktionen

...

In Java



# Refactoring towards monadic transactions

...

In Java



# About Me

## Gregor Trefs

32 years old

Senior software engineer

Organizer of [@majug](#)

Founder of [@swkrheinneckar](#)

twitter/github: [gtrefs](#)



# About You

Who knows what  
a function is?  
a lambda expression is?  
a higher order function is?  
a monad is?

```
public void save(T entity) {
    beginTransaction();
    entityManager.persist(entity);
    commitTransaction();
}

private void beginTransaction() {
    try {
        entityManager.getTransaction().begin();
    } catch (IllegalStateException e) {
        rollBackTransaction();
    }
}

private void commitTransaction() {
    try {
        entityManager.getTransaction().commit();
    } catch (IllegalStateException | RollbackException e) {
        rollBackTransaction();
    }
}
```

**A typical repository**

Move a scattered and/or repeated responsibility into one single method

Execute around method pattern

```
public void update(int id, Consumer<T>... updates) throws Exception {
    T entity = find(id);
    transactional(em -> Arrays.stream(updates).forEach(up -> up.accept(entity)));
}

public void remove(int id) {
    transactional(em -> em.remove(find(id)));
}

private void transactional(Consumer<EntityManager> action) {
    try {
        entityManager.getTransaction().begin();
        action.accept(entityManager);
        entityManager.getTransaction().commit();
    } catch (RuntimeException e) {
        entityManager.getTransaction().rollback();
        throw e;
    }
}
```

## Execute around method pattern

Concise code

Execute around method pattern



Reduces the risk of incorrect transaction  
management

Execute around method pattern

A function should only return nothing if it does nothing. Any other observed effect is a side effect.

Execute around method pattern

Transaction execution is the immediate side  
effect

Execute around method pattern

Difficult code reuse: update and remove in the same transaction?

Execute around method pattern

Higher order

First order

```
int compute(int i, Function<Int, Int> f){  
    int increased = i + 1;  
    return f.apply(increased);  
}
```

First concern

Second concern

Higher order functions

A means to describe the transaction and  
delay its execution

Higher order functions

```
public T convert(int id, UnaryOperator<T> converter){
    Function<EntityManager, T> find = em -> em.find(entityType, id);
    Function<EntityManager, T> transaction = transactional(find.andThen(converter));
    return transaction.apply(entityManager);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
    return em -> {
        try {
            em.getTransaction().begin();
            final U result = action.apply(em);
            em.getTransaction().commit();
            return result;
        } catch (RuntimeException e) {
            em.getTransaction().rollback();
            throw e;
        }
    };
}
```

## Higher order functions

Compose descriptions without the burden of  
side effects

Higher order functions



Execute the description by applying it to the  
entity manager

Higher order functions

The actual database is determined upon  
execution

Higher order functions

```
public Function<EntityManager, T> convert(int id, UnaryOperator<T> converter){
    return transactional(find(id).andThen(converter));
}

public Function<EntityManager, T> find(int id){
    return em -> em.find(entityType, id);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
    return em -> {
        try {
            em.getTransaction().begin();
            final U result = action.apply(em);
            em.getTransaction().commit();
            return result;
        } catch (RuntimeException e) {
            em.getTransaction().rollback();
            throw e;
        }
    };
}
```

## Higher order functions

Type Function does not convey the information whether a transaction is executed during application

Higher order functions

```
public Function<EntityManager, T> transactionalConvert(int id, UnaryOperator<T> converter){
    return transactional(transactional(find(id)).andThen(converter));
}

public Function<EntityManager, T> find(int id){
    return em -> em.find(entityType, id);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
    return em -> {
        try {
            em.getTransaction().begin();
            final U result = action.apply(em);
            em.getTransaction().commit();
            return result;
        } catch (RuntimeException e) {
            em.getTransaction().rollback();
            throw e;
        }
    };
}
```

## Higher order functions

Function

Function



A Transaction type to put an action in, to  
combine it with others and to run it

Function



Towards a transaction monad

Is type `Transaction` a monad?

Towards a transaction monad

Functional languages are based on the  
lambda calculus

Brief background about functional languages



# Applying conversions on expressions

$\lambda x . x$

Brief background about functional languages

How to integrate side effects and stay pure?

Brief background about functional languages

Don't: Lisp (Clojure, Scheme), Standard ML  
(println (read-line))

Brief background about functional languages

“Or I could use a monad” -- Philip Wadler

Brief background about functional languages

“‘In order to understand monads you first need to learn category theory’ is like saying ‘In order to understand Pizza you first need to learn Italian.’” -- Mario Fusco (Italian)

Brief background about functional languages

A monad of type  $M$  represents some computation: I/O, potential absent values, values available in the future, lists, etc.

A brief and mostly incorrect introduction to monads

A function to turn a value into a computation  
that produces the value  
 $M\langle T \rangle$  of (T value)

A brief and mostly incorrect introduction to monads

A function to combine computations

```
M<U> flatMap(M<T> m, Function<T, M<U>> f)
```

A brief and mostly incorrect introduction to monads



```
M<String> hello = of("hello");  
  
Function<String, M<String>> world = str -> of(str + "World");  
  
M<String> helloWorld = flatMap(hello, world);
```

**A brief and mostly incorrect introduction to monads**

A monad is a tuple  $(M, \text{of}, \text{flatMap})$

A brief and mostly incorrect introduction to monads

Monad laws describe how operations relate to each other and, thus, make reasonable assumptions about their behavior

A brief and mostly incorrect introduction to monads

Identity laws guarantee that the `of` function just puts the value into the computation and does not manipulate it

A brief and mostly incorrect introduction to monads

Left side



```
flatMap(of(v), f) == f.apply(v)  
flatMap(m, v -> of(v)) == m
```

Right side



A brief and mostly incorrect introduction to monads

The associativity law guarantees that function composition holds across the combination of computations

A brief and mostly incorrect introduction to monads

```
flatMap(flatMap(m, f), g) ==  
flatMap(m, v -> flatMap(f.apply(v), g))
```

A brief and mostly incorrect introduction to monads

Laws do not provide a mental model what a monad is or what a monad means

A brief and mostly incorrect introduction to monads



We are used to perceive interfaces as a generalization of specific representations

A brief and mostly incorrect introduction to monads

For example: Interface `List` is a generalization of `ArrayList` and `LinkedList`

A brief and mostly incorrect introduction to monads

Monad does not generalize one type or  
another

A brief and mostly incorrect introduction to monads

A type is monadic if it has operations which  
satisfy the laws

A brief and mostly incorrect introduction to monads

The monad operations are often just a small fragment of the full API for a given type that happens to be a monad

A brief and mostly incorrect introduction to monads

The monad contract does not specify what is happening between the lines, only that whatever is happening satisfies the laws

A brief and mostly incorrect introduction to monads

# CompletableFuture Stream

```
completedFuture("hello").thenCompose(v -> completedFuture(v + "world"));  
completedFuture("hello").thenComposeAsync(v -> completedFuture(v + "world"));  
Stream.of("hello").flatMap(v -> Stream.of(v + "world"));
```

## Monads in Java



Type Transaction is monadic\*

Towards a transaction monad

```
userRepository.convert(10, clone).run(entityManager);
final UnaryOperator<User> clone = user -> new User(user.getName(), user.getEmail());

public Transaction<T> convert(int id, UnaryOperator<T> converter){
    return find(id).flatMap(entity -> Transaction.of(converter.apply(entity)));
}

public Transaction<T> find(int id) {
    return findById(id).apply(entityClass);
}

private Function<Class<T>, Transaction<T>> findById(int id) {
    return clazz -> Transaction.of(em -> em.find(clazz, id));
}
```

## Towards a transaction monad

Given the capability of describing a transaction before execution, a simple set of CRUD transactions can be factored in an own interface

Towards a transaction monad

```
public interface CrudTransactions<T> {  
  
    default Transaction<Void> saveEntity(T entity) {  
        return transactional(em -> em.persist(entity));  
    }  
  
    default Transaction<Void> removeEntity(T entity) {  
        return transactional(em -> em.remove(entity));  
    }  
  
    default Transaction<Void> transactional(Consumer<EntityManager> action){  
        return Transaction.withoutResult(action);  
    }  
  
    default Function<Class<T>, Transaction<T>> findById(int id) {  
        return clazz -> Transaction.of(em -> em.find(clazz, id));  
    }  
}
```

## Towards a transaction monad

```
public class Repo<T> implements EntityRepository<T>, CrudTransactions<T> {  
    // some code is left out  
    public Transaction<T> find(int id) {  
        return findById(id).apply(entityClass);  
    }  
  
    public Transaction<T> convert(int id, UnaryOperator<T> conv){  
        return find(id).flatMap(entity -> Transaction.of(conv.apply(entity)));  
    }  
  
    public Transaction<Void> save(T entity) {  
        return saveEntity(entity);  
    }  
  
    public Transaction<Void> remove(int id) {  
        return find(id).flatMap(this::removeEntity);  
    }  
}
```

**Towards a transaction monad**

Build up reusable vocabularies to talk to the  
databases

Towards a transaction monad

UserDetails implements  
Read<User>, Count<User>

Towards a transaction monad

# Code of Transaction<T>

Towards a transaction monad



Knowing a type is monadic, let us reason  
about its behavior

Benefits of monads

For example: Exploit the associativity law and rearrange the function chaining

Benefits of monads

```
completedFuture("hello")
  .thenCompose(v -> completedFuture(v + "world"))
  .thenCompose(v -> completedFuture(v + "2017"));

completedFuture("hello").thenCompose(v ->
  completedFuture(v + "world")
  .thenCompose(w -> completedFuture(w + "2017")))
);
```

## Benefits of monads

Java is a poor tool for monads

Benefits of monads

```
describe("Combination of crud methods"){
  it("should combine findById with update"){
    val em = ???
    val user = new User("Test", "test")
    findAndUpdate(user.getId, classOf[User], _.setEmail("mail")).run(em)

    def findAndUpdate(id:Int,clazz:Class[User],updates:Consumer[User]) = for {
      entity <- findById(id)(clazz)
      update <- updateEntity(entity, updates)
    } yield update

  }
}
```

## Benefits of monads

Monads are a part of a solution to a problem  
that never existed in Java

Benefits of monads

Though monadic types help us dealing with side effects in a predictable way

Benefits of monads



TL;DR

A monad

is a triple (M, of, flatMap)

adheres to laws

is a *self-containing* interface

is not well supported in Java

Transaction

is a monadic type

defines reusable transactions

is a specialization of Reader





# The End

Questions?

## Contact

[Gregor.Trefs@gmail.com](mailto:Gregor.Trefs@gmail.com)

[linkedin.com/in/gregor-trefs](https://www.linkedin.com/in/gregor-trefs)



# Literature

and links

- My blog
  - <https://gtrefs.github.com>
- FP in Scala
  - Paul Chiusano
  - Rúnar Bjarnason
- The essence of FP
  - Philip Wadler
- Background picture
  - by John Salzarulo