Composability and the actor model

Jan Macháček[*†]

April 19, 2018

Abstract

1 Composition

Composition is a way of combining several smaller units of functionality to build larger ones. It does not prescribe any particular programming style: calling other functions from a functions is compisition, so is dependency injection in traditional OO world.

To demonstrate, consider an order processing system that fulfils the order contained within a request by decoding the order data from the request, calculates discounts and books the best delivery method, before finalising the order and encoding it into a response can compute the discount and delivery method at the same time (to use a loose term for now), though the other operations must be sequential; see Figure 1.

$$decode \rightsquigarrow \left\{ \begin{array}{c} delivery \\ discount \end{array} \right\} \rightsquigarrow finalise \rightsquigarrow encode$$

Figure 1: Order fulfilment

This type of processing is a typical result of decomposing the *order fulfilment* into its steps. The decomposition and the algorithm assume that the entire fulfilment operation is atomic and that its effects on the world are completely isolated until the *encode* operation successfully completes.

1.1 Serial and pure implementation

It is easiest to start exploring the problem using an imperative pseudo-code in Listing 1.

```
order = decode(request)
delivery = delivery(order)
discount = discount(order)
fulfiled = finalise(order, delivery, discount)
response = encode(fulfiled)
```

Listing 1: Fulfilment implementation I

This code is an accurate representation of the process when the *decode* , *delivery* , *discount* , *finalise* , and *encode* functions are pure: the functions' results depend only on the values given as parameters; their types are $I \Rightarrow O$ for the appropriate types I and O . (The type of the *decode* function, for example, is $Request \Rightarrow Order$). In this order fulfilment system, the "business" functions need to interact with the world to arrive at their result.

In some very popular languages (cite: Java, C#) it is not possible to define plain functions: every function needs to be part of a class[1]. Even though it is possible to define ***static*** method in these languages, where the method can be invoked without creating an instance of the class–and thus treating the class containing the static method as a namespace–it is usual to define and use interfaces that define the beaviour[2]. This yields code in Listing 2.

```
interface OrderCodec {
  Order decode(Request r);
  Response encode(Order o);
}
interface DeliveryBookingService {
  Delivery book(Order o);
}
interface DiscountCalculatorService {
  Discount calculate(Order o);
}
interface OrderFinalisationService {
  Fulfiled finalise(Order o,
                    Delivery del,
                    Discount dis);
}
```

Listing 2: Fulfilment OO implementation I

This is almost mechanical wrapping of the naked functions into interfaces; the implementations of these interfaces contain the code that the naked functions contained. The code from Listing 1 then becomes part of another class in Listing 3.

```
interface Fulfilment {
  Response fulfil(Request request);
}
class FulfilmentImpl implements Fulfilment {
  OrderCodec oc;
  DeliveryBookingService dbs;
  DiscountCalculatorService dcs;
  OrderFinalisationService ofs;

  Response fulfil(Request request) {
    var order = oc.decode(request);
    var delivery = dbs.book(order);
    var discount = dcs.calculate(order);
    var fulfiled = ofs.finalise(
      order, delivery, discount);
    return oc.encode(fulfiled);
  }
}
```

Listing 3: Fulfilment OO implementation II

[1] ***class*** means template for instances.

The *Fulfilment* *composes* the sub-components (*OrderCodec* , *DeliveryBookingService* and others) into a larger unit. The interface *Fulfilment* hides all the complexity of the processing; the implementation *FulfilmentImpl* delegates the work to its sub-components. The code in Listing 3 omits the construction of these components. The usual approach is to avoid having the *OrderHandlerImpl* be responsible for creating its dependencies to keep to the single-responsibility principle; so the *OrderHandlerImpl* must be given these dependencies at the point of its construction. It becomes

```
interface Fulfilment {
  Response fulfil(Request request);
}
class FulfilmentImpl implements Fulfilment {
  private final OrderCodec oc;
  private final DeliveryBookingService dbs;
  private final DiscountCalculatorService dcs;
  private final OrderFinalisationService ofs;

  FulfilmentImpl(
    OrderCodec oc,
    DeliveryBookingService dbs,
    DiscountCalculatorService dcs,
    OrderFinalisationService ofs) {
    this.oc = oc;
    this.dbs = dbs;
    this.dcs = dcs;
    this.ofs = ofs;
  }

  Response fulfil(Request request) { ... }
}
```

Listing 4: Fulfilment OO implementation II

In large systems, the construction of all components–each with its own chain of dependencies–becomes tedious. The tedious work of manual construction of all components might even slow down any refactoring that affects the construction graph. An alternative approach is to use dependency injection pattern: a DI framework maintains an environment of all constructed / constructible components. When a new component needs to be added into this environment, the framework satisfies its dependencies (expressed through constructor parameters, setters, or using some other mechanism), registeres the newly created component in the environment, and returns the registered instance. This way, the code that makes up the components looks like plain Java, but the components are implicitly tied to this environment maintained by the DI framework. This low-level composition is comfortable; it is also accurate representation of our initial decomposition if the methods in the components are still pure. The *FulfilmentImpl. fulfil* remains pure if the components injected at the point of construction do not change throughout the life of the system and if their methods are pure.

1.2 Effects

Useful systems exist only for their side-effects.

The *delivery* function (or the *DeliveryBookingService .book* method) needs to call various couriers' APIs to select & book the best one; the *discount* function needs to query an affiliate database for possible price reduction; the *finalise* function needs to make further external API calls and database operations. The only functions that remain as pure are *decode* and *decode* . If the implementation uses a DI framework, the database connection or the API clients are injected into the components; the injected dependencies do not change, but their methods are not pure. Many programming languages & runtimes use exceptions as a way to keep the functions' types intact, but to provide a mechanism to indicate that the purity abstraction has leaked. In Java, one might implement the *delivery* function as *Delivery book(Request r)* ***throws*** *DeliveryException* , where the *DeliveryException* is a root of an inheritance hierarchy of more detailed exceptions. The callers are then responsible for handling any raised exceptions. The usual code is in Listing 5.

```
DeliveryBookingService dbs = ...;
Order o = ...;
try {
  dbs.book();
} catch (DeliveryException ex) {
  oops, not booked; maybe we retry?
}
```

Listing 5: Exceptions

The devil is in the detail: the code in Listing 5 assumes that the *DeliveryBookingService.book* is isolated and atomic; an exception arising from the call means that everything that the *DeliveryBookingService.book* has done has been undone. Abandoning inheritance hierarchy of exceptions provides partial solution. Remove the *DeliveryException* superclass and have the method throw–for example– *DeliveryAPIUnavailable* , *DeliveryRejected* , and *DeliveryBookingTimeout* . The *catch* block becomes longer, but has the opportunity to handle specific error cases. Unfortunately, in Java and C#, all exceptions must implement the *Throwable* interface. This, combined with the fact that writing long *catch* blocks is tedious, leads to shortcuts. Instead of exhaustive error handling in Listing 6...

```
DeliveryBookingService dbs = ...;
Order o = ...;
try {
  dbs.book();
} catch (DeliveryAPIUnavailable ex) {
  Retryable, nothing to undo
} catch (DeliveryRejected ex) {
  Not retryable, but nothing to undo
} catch (DeliveryBookingTimeout ex) {
  Has the other system received the request and
  failed to respond, or has the request not reached
  the other system at all?
}
```

Listing 6: Exceptions

...the code [ab]uses the root exception hierarchy imposed by the language and ends up as Listing 7.

```
DeliveryBookingService dbs = ...;
Order o = ...;
try {
  dbs.book();
} catch (Exception ex) {
  An unexpected error has occurred. Do you feel lucky?
}
```

Listing 7: Exceptions

Unfortunately, exceptions are not sufficient to express the richness of the possible failures that can happen with code that relies on side-effects, and the abstraction of purity in the world of side-effects leaks. The immediate and hard errors can be expressed with exceptions: the *connection denied* or *illegal argument exception* are expressive enough and can be raised without any delays. Detecting and raising a *request timeout* exception takes more time; during this time, the thread of execution must be blocked.

1.3 Concurrency and parallelism

The world is concurrent, and parts of the world can be parallel. Concurrency means many different things happening at the same time: think different users making requests to the order fulfilment system. Parallellism means splitting larger unit of work into smaller independent units of work and then combining their result: think vector dot product running on a GPU, where each multiplication is independent of the other and runs on a different GPU core.

When two concurrent processes do not need to communicate with each other they proceed in parallel; when two concurrent processes have to communicate, it is necessary to understand the mechanism of the communication to be able to build error-free programs. A typical error in concurrent programs is unsynchronised access to a shared state: the processes *oneStepForward* and *oneStepBack* both access the same shared *state* in Listing 8.

```
The shared mutable state
state = 0

The processes that mutate the state
oneStepForward = state += 1
oneStepBack = state -= 1

Threads provide mechanism to run the two processes concurrently
startThread(oneStepForward)
startThread(oneStepBack)

The parent process collects the result
if state ≠ 0 then output(state)
```

Listing 8: Shared mutable state

The value passed to the *output* call is not deterministic; and it can even be zero, because the *state ≠ 0* condition evaluates to ***true***, but then one of the threads mutates the *state*, but the code is already following the “then” branch. The concurrent processes need to synchronise access and mutation to the shared state. (Viz Listing 9)

```
The shared mutable state
state = 0
The lock that only allows one process to enter
lock = makeLock

The processes acquire and release the lock
oneStepForward = with(lock) { state += 1 }
oneStepBack = with(lock) { state -= 1 }

...

The parent also locks on the state
with(lock) {
  if state ≠ 0 then output(state)
}
```

Listing 9: Locked shared mutable state

The value of *state* in program in Listing 9 isn't deterministic because the order order in which the concurrent processes *oneStepForward* and *oneStepBack* are selected to run isn't determined, but the locks ensure that the state doesn't change while one particular process is using it. The size of the program is small enough to be able to reason about the concurrent processes; just as importantly, the program is running on a single machine and state is in memory on the same machine. A single machine provides almost leak-free abstraction of reliable computation and immediate memory access.

This still isn't a problem until two processes need to communicate to reach agreement on some state, or to cooperate in computation.

Changing the type O to higher-kinded type $\mathbb{F}$ provides a way to express that the returned value is not the raw value of type O, but that it is wrapped in some container $\mathbb{F}$. For $\mathbb{F} = Future$, *decode(Request): Order* becomes *decode(Request): Future⟦Order⟧*, which expresses the fact that the result of applying these functions to the input does not yield the result immediately; it *starts* the computation and returns the conainer $\mathbb{F} = Future$ which will hold the computed value. Viz Listing 10.

```
order = decode(request)
(delivery, discount) ← delivery(order)
                     ⋏ discount(order)
fulfiled ← finalise(order, delivery, discount)
response = encode(fulfiled)
```

Listing 10: Fulfilment implementation II

The only difference between the two listings is the changing of assignment operator = to the sequence / shove operator ← ; and the usage of the ⋏ operator to “zip” two $\mathbb{F}$s together: $\mathbb{F}[\![A]\!] \curlywedge \mathbb{F}[\![B]\!] \Rightarrow \mathbb{F}[\![(A,B)]\!]$. This changes the type of *order* to *Future⟦Order⟧*, the type of *delivery* to *Future ⟦Delivery⟧*, . . . , the type of *response* to *Future⟦Response⟧* . It is possible to *compose* the components that deal with decoding, booking delivery, computing a discount, finalising, and encoding to a larger component of type *Request ⇒ Future⟦Response⟧*.

The *Future* can succeed or fail, making the code in Listing 10 feel right. The code does not handle any failures

The fulfilment operation isn't instant, and the world does not stop while it is processing. If the entire fulfilment operation could be done in the same memory and by only affecting that memory, it would be possible to use software transactional memory[4]. If the decomposed steps in the fulfilment process change the world (by initiating network communication, by printing labels, etc.) they cannot be simply retried: is it OK to make a delivery booking twice?; is it acceptable to print the packaging label multiple times? Deduplicating non-idempotent requests encounters storage limits: any deduplicating code can only deduplicate on a fixed number of requests. Locking reduces the throughput of a system as the number of locks grows: locking code has to be able to old locks.

Even if isolation and atomicity were achievable in the context of this system, the *delivery* booking system might not be able to participate in any form of transaction. Worse still, the *delivery* booking step might have returned a successful booking, but by the time the *finalise* step executes, the delivery booking becomes invalid.

encode can fail.

DeliveryEstimate: Order => Days

Discount: Order => Days

Fulfilment(deliveryEstimate, discount): Order => Fulfilment = order => deliveryEstimate(order) and discount(order)

Pure actors do not compose.

2 Ramblings & notes

Asynchrony and concurrency is difficult; it is tempting to dismiss it as *too complicated* for applications that "simply" take a request and produce a response, where the work to produce the response involves simple data transformations and straight-forward logic.

```
The actual business logic that computes the delivery estimate for
the given item
deliveryEstimate(item: Item): Days

Mechanics for transforming between the request and response and
the application's data model
fromRequest(request: Request): Order
toResponse(estimate: Days): Response

The main handling code that computes the delivery estimate for
the order in the request, producing the estimate in the response
handle(request: Request): Response =
  items    ← parseItems(request)
  estimate ← 0
  for (item in items) {
    e ← deliveryEstimate(item)
    if (e > estimate) estimate = e
  }
  return toResponse(estimate)
}
```

Listing 11: Delivery estimate

If the *deliveryEstimate* function is *pure* (its result depends only on the given parameter and nothing else) making the *handle* function also pure, then it would appear that this code is a perfect candidate to use a thread-per-request model and not worry about any concurrency at all. The answer depends on how the application gets the *Request* and where the *Response* ends up[2], how many requests the application handles concurrently, what response time guarantees the application makes, and on the underlying implementation of the thread of execution.

Given N_t number of threads and N_r, there are the following scenarios:

* $N_r \ll N_t$ and low response time critical

* $N_r \leq N_t$

* $N_r > N_t$

* $N_r \gg N_t$

Thread-per-request model works if the delivery estimate module is used by another module that is also pure (see Listing 12)

```
The delivery estimate module with its specific handle function
module DeliveryEstimate =
  handle(request: Request): Response

The discount module with its specific handle function
module Discount =
  handle(request: Request): Response

toResponse(r1: DeliveryEstimate.Response,
           r2: Discount.Response): Response
handle(request: Request): Response =
  r1 ← DeliveryEstimate.handle(request)
  r2 ← Discount.handle(request)
  return toResponse(r1, r2)
```

Listing 12: Delivery and discount

3 Actors

There are many actor frameworks and toolkits[1, 3, 6, 5]. Some toolkits make it very difficult to compose the behaviour of the actors[1, 5], other toolkits make it much easier[3, 6]. I consider the actor model as a way to *decompose* a system into self-contained units of functionality and state. I consider these units of functionality and state to be the smallest deployable entities; consequently, the communication between two actors must take form of messages travelling over a boundary that introduces latency and the risk of message loss. Given this definition, it does not make sense to require convenient general composition mechanism for general actors. Such composition mechanism is an abstraction that must attempt to hide the underlying latency and the risk of message loss[3].

References

[1] *Akka*. https://akka.io.

[2] Rob Harrop Jan Machacek. *Pro Spring*. TODO: Apress, TODO: 2012.

[3] *Scalaz Actors*. https://github.com/scalaz/scalaz/tree/series/7.3.x/concurrent/src/main/scala/scalaz/concurrent.

[4] *Software transactional memory*. https://wiki.haskell.org/Software_transactional_memory.

[5] *Thespian*. https://bitbucket.org/alinabi/thespian.

[6] *Transient*. https://github.com/transient-haskell/transient.

[2] I am not going to follow the rabbit hole of layering in more and more pure functions: programs run to interact with the world by the means of impure I/O. However, I do not dismiss purity or the ability to track the spread of arbitrary effects such as I/O.

[3] Consider the *Network File System*, which abstracts over unreliable networks to provide the illusion that a filesystem on a remote machine is the same as the filesystem on the local machine. The abstraction *leaks* when the network is slow or lossy, when a lot of small files are being accessed over the network, when a different application generates a lot of traffic on the NIC that also handles the NFS mount, etc.