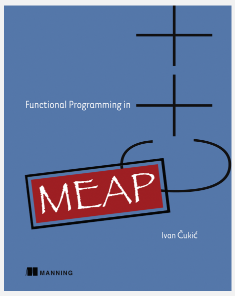

Functional Design for Concurrent Systems

by Ivan Čukić

The biggest problem in software development is handling complexity. Software systems tend to grow significantly over time and they quickly outgrow the original designs. When it turns out that the features that need to be implemented collide with the design, we must either re-implement significant portions of the system or introduce horrible quick-and-dirty hacks to make things work.

This problem with complexity becomes more evident in software which has different parts that execute concurrently – from the simplest interactive user applications, to network services and distributed software systems.

> "A large fraction of the flaws in software development are due to programmers not fully understanding all the **possible states** their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are **greatly amplified**, almost to the point of panic if you are paying attention."
> – John Carmack - In-depth: Functional programming in C++

Most of these problems come from the entanglement of different system components. Having separate components that access and mutate the same data requires synchronizing said access. This synchronization is traditionally handled with mutexes or similar synchronization primitives, which works, but it introduces significant scalability problems and it kills concurrency.

One approach to solving the problem of shared mutable data is not having any mutable data whatsoever. But there's another option – to have mutable data, but never to share it.

The actor model – thinking in components

In this article, we'll see how to design the software as a set of isolated separate components. We'll first need to discuss this in the context of object-oriented design in order to later see how we can make it functional.

When designing classes, we tend to create getter and setter functions – getters to retrieve information about an object, and setters to change attributes of an object in a valid way which won't violate the class invariants.

Many object-oriented design proponents consider this approach to be contrary to the philosophy of OO. They tend to call it procedural programming because we still think in algorithm steps, and the objects serve as containers and validators for the data.

> Step one in the transformation of a successful procedural developer into a suc-

cessful object developer is a lobotomy.
– David West - Object Thinking

Instead, we should stop thinking about what data an object contains, and think about what it can do. As an example, consider a class which represents a person. The way we'd usually implement it is to create getters and setters for the name, surname, age and other attributes. Then, we could do something like:

```
douglas.set_age(42);
```

And this shows the problem. We've designed a class to be a data container, instead of designing it to behave like a person. Can we force a human being to be 42 years old? We can't. We can't change the age of a person, and we shouldn't design our classes to allow us to do this.

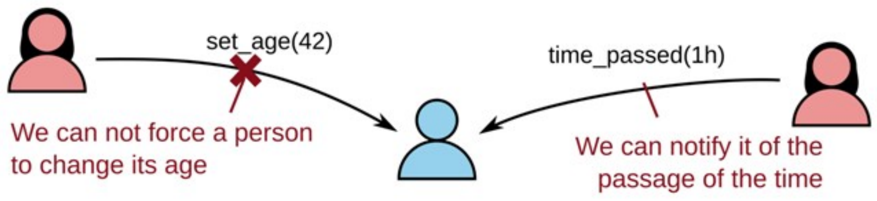
set_age(42)
We can not force a person to change its age
time_passed(1h)
We can notify it of the passage of the time

Figure 1: We can't set the attributes on real-life objects. We can send them messages, and let them react to them.

We should design the classes with a set of actions or tasks they can perform, and then add the data necessary to implement those actions. In the case of the class which models a person, instead of having a setter for the age, we'd need to create an action to tell the person that some time has passed, and the object should react appropriately. Instead of `set_age`, the object could have a member function `time_passed` like so:

```
void time_passed(const std::chrono::duration& time);
```

When notified that the specified time has passed, the person object can increase its age, but also perform other related changes. For example, if this is relevant to our system, the person's height could be changed, the hair color etc. as the result of the person getting older. Therefore, instead of having getters and setters, we'd only have a set of tasks that that the object knows how to perform.

> Don't ask for the information you need to do the work; ask the object that has the information to do the work for you.
> – Allen Holub

If we continue to model the person object after real-life people, we'll also come to a realization that multiple-person objects shouldn't have any shared data. Real people share data by talking to each other; they don't have shared variables which everyone can access and change. This is exactly the idea behind actors. In the actor model, actors are completely isolated entities which share nothing, but which can send messages to one another. At a minimum an actor class should have a way to receive and send messages.

Traditionally, actors can send and receive different types of messages, and each actor can

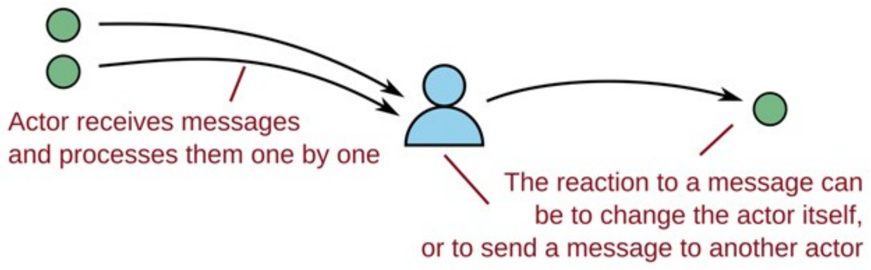
Actor receives messages
and processes them one by one
The reaction to a message can
be to change the actor itself,
or to send a message to another actor

Figure 2: An actor is an isolated component that can receive and send messages. It processes the messages serially, and for each message it can change its state or behavior, or it can send a new message to another actor in the system.

choose which actor to send the message to. Also, the communication should be asynchronous.

C++ Actor Framework

You can find a complete implementation of the traditional actor model for C++ at http://actor-framework.org/ which you can use in your projects.
The C++ Actor Framework has an impressive set of features. The actors are lightweight concurrent processes (much more lightweight than threads) and it's network-transparent, meaning that you can distribute your actors over multiple separate machines and things will work without the need to change your code.
Although traditional actors aren't easily composed, they can easily be made to fit into the design we'll cover in this article.

We're going to define a more rudimentary actor compared to actors as specified by the actor model and actors in the C++ Actor Framework because we want to focus more on the software design than on implementing a true actor model. Although the design of actors presented in this article differs from the design of actors in the traditional actor model, the concepts which will be presented are applicable even with the traditional actors. We're going to design our actors:

- To be typed actors – actors that can receive only messages of a single type, and send messages of a single (not necessarily the same) type. If we need to support multiple different types for input or output messages, we can use std::variant or std::any.
- Instead of allowing each actor to choose to whom to send a message, we'll leave this choice to an external controller which will allow us to compose the actors in a functional way. The external controller schedules which sources an actor should listen to.
- We'll leave it up to the external controller to decide which messages should be processed asynchronously and which not

Most software systems nowadays use or implement some kind of event loop which can be used to asynchronously deliver messages, and we won't concern ourselves here with implementing such a system – we'll focus on the software design which can easily be adapted to work on any event-based system.

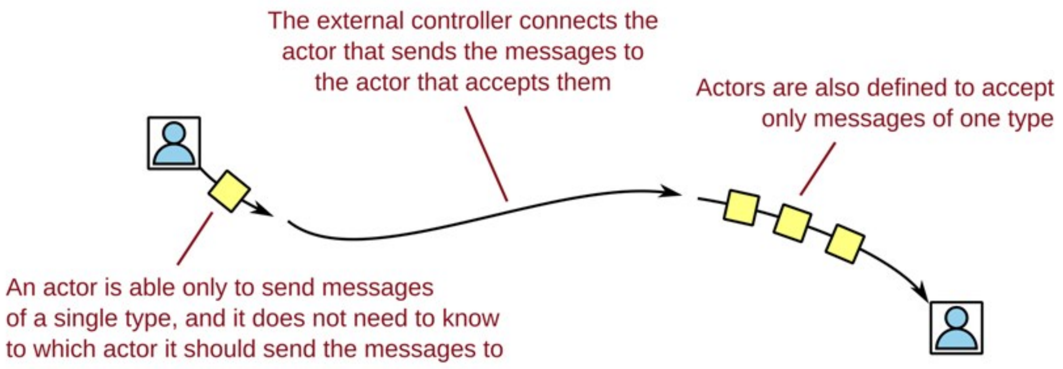
The external controller connects the actor that sends the messages to the actor that accepts them
Actors are also defined to accept only messages of one type
An actor is able only to send messages of a single type, and it does not need to know to which actor it should send the messages to

Figure 3: We'll use simplified typed actors that don't care who sends the message to whom because this is left to an external controller.

```
template <typename SourceMessageType,
          typename MessageType>
class actor {
public:
    using value_type = MessageType;

    void process_message(SourceMessageType&& message);

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit);

private:
    std::function<void(MessageType&&)> m_emit;
};
```

An actor can receive messages of one, and send messages of another type

Defines the type of the message the actor's sending, allowing us to check it when we need to connect the actors to one another

Handles when a new message arrives

Sets the m_emit handler which the actor calls when it wants to send a message

With this interface, we're clearly stating that an actor knows only how to receive a message, and how to send a message onwards. It can have as many private data as it needs to perform its work, but none of it should ever be available to the outside world. Because the data can't be shared, we have no need to synchronize it.

It's important to note that there can be actors that only receive messages (usually called *sinks*), actors that only send messages (usually called *sources*) and general actors that do both.

That's all for now. If you're hungering for more new insights into how you can leverage functional programming techniques to write better C++ code, have a look at the first chapter of Functional Programming in C++ and see this slide deck.