Patterns and practices

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- State machines
- Entity/component
- Option with higher order function accessors

Introduction

Introduction

Agenda

- In this lecture we perform a code-centered review of the topics seen so far
- We will see (by coding them) a series of examples of polymorphism, generics, and higher order functions

State machines

State machines

Moving parts

- State machines are all based on the implementation of the `StateMachine` interface, with methods:
  - `Step`, which returns `true` when it is done and `false` when it is still running
  - `Reset`, which resets the state machine to its initial state
- Concrete implementations of `StateMachine` are (just for this example):
  - `Wait`, which waits for a given amount of time
  - `Repeat`, which repeats forever the state machine it gets as argument
  - `SayHello`, which prints hello on the screen just once

Entity/components

Entity/components

Moving parts

- An entity/component system is based on the composition of an entity by means of multiple generic components:
- The entity in our example is a `Car`, which features a series of components:
  - `FuelTank`, which is an interface
  - `Engine`, which is an interface
  - `Wheels`, which is an interface
- When creating a concrete `Car`, we must pass an instance of a concrete implementor of the above interfaces
- The `Car` simply connects the components, but has no idea what they do precisely

Option with higher order function accessors

Option with higher order function accessors

Moving parts

- An option data type is a wrapper around a value of a generic type `T`, which might (or might not) be absent
- The `Option<T>` interface has only one method:
  - `Visit`, which takes as input two functions: one to process the value, one to provide a fallback otherwise
- There are only two concrete implementations of `Option<T>`:
  - `Some<T>`, which contains a value of type `T`
  - `None<T>`, which contains no value of type `T` and acts as a sort of strongly typed `null` value

Conclusion

Conclusion

Looking back

- Polymorphism makes it possible to pass different data types to other contexts, as long as the conversion is safe
- Generics make it possible to define a class once, but use it with multiple types as arguments
- Their combination makes it possible to reach amazing levels of abstraction, but require careful thought to be used correctly
- Use design (and UML-style reasoning) like violence: if it does not solve the problem, just use more

This is it!

The best of luck, and thanks for the attention!