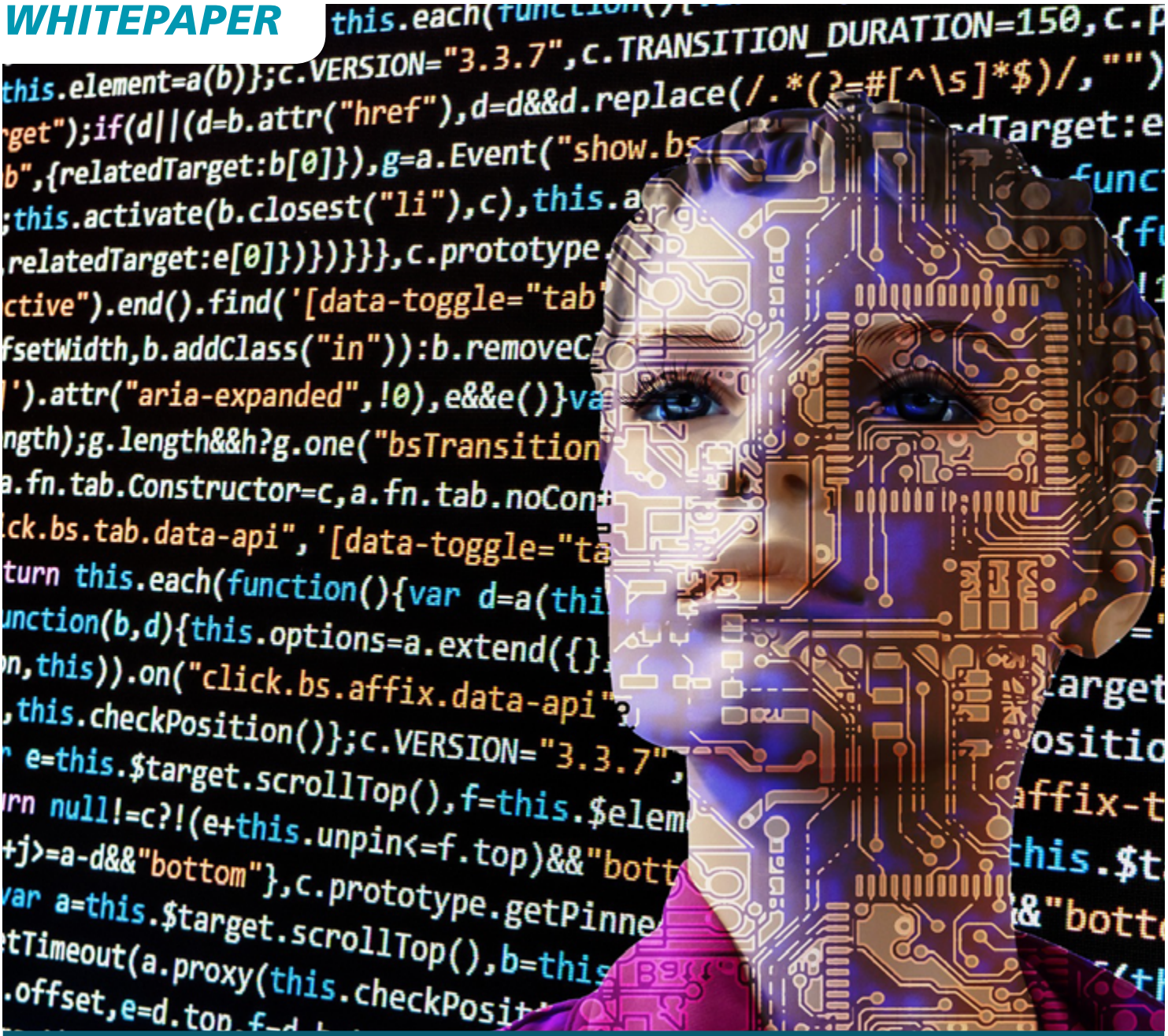
WHITEPAPER



Start Increasing Your Software Productivity: Model Driven Technologies, The State Pattern, and Dezyne

Ewald de Bruijn, Jasper Premchand, Julien Schmaltz, ICT Group, September 2019

ICT smarter solutions

Abstract: The state pattern provides software designers with an implementation of state machine such that states and transitions are separated from the actual actions performed by a component. Dezyne is an environment supporting formal verification and code generation for the design of complex interactions between state machines. In this paper, we show a systematic approach to introduce Dezyne generated code in a state driven component built using the State Pattern. This first step towards the new standard in software Engineering is relatively easy and brings about many benefits, including, increase in code readability, state charts visualisation, synchronisation of documentation with code, and error-free code generation.

1 INTRODUCTION

Innovation is driven by software. To support the further development of software intensive systems, a lot of software must be written in a short amount of time and with good quality. Major issues preventing the rapid improvement of these systems include the cost of ensuring software quality and the lack of software designers. It is very difficult to recruit the required personnel. Developing good quality software requires much effort. For instance, a lot of time is spent in keeping documentation and code synchronised. Also, when the system gets complex, it becomes very difficult to still see the big picture. Migration and other re-factoring tasks become harder and harder.

Model Driven Technologies (MDT) constitute promising solutions to help software intensive companies to be more productive. From models code is automatically generated. This generation is error-free. Many artefacts can be automatically generated from models: documentation, graphical representations like state charts or sequence diagrams. For many companies, a major challenge is how to access these new model driven technologies and gain their benefit. The question is how to introduce these new technologies while staying open for usual business. This is exactly the challenge addressed by this paper.

At ICT, our objective is to guide customers in adopting and deploying this innovative software Engineering solutions. We want to guide them through several steps, each step improving their software Engineering process and making them capable to sustain their market position and boost their innovation capabilities.

In this paper, we describe one of these steps, namely, the replacement of handwritten code with code generated from models in the **Context** of state machines implemented following the state pattern. Models are written using the Dezyne language developed by Verum Software Tools[1]. After introducing the necessary background, including Dezyne, the state pattern and a running example, we show our approach to replace handwritten code for states and transitions with code generated from Dezyne models. The Visual Studio projects and the Dezyne models can be found on-line[2].

2 BACKGROUND

2.1 A running example: a simple Engine

As a running example, we consider a simple **Engine**. The **Engine** can be in two states: either it is **ON** or it is **OFF**. The **Engine** initially is **OFF.** Users can turn the **Engine** on and off using commands **StartEngine** and **TurnOffEngine**. For some reasons (for instance, safety or mechanical aspects), there are some restrictions about when commands can be executed:

- A **StartEngine** command while the **Engine** is **ON** will damage the **Engine**.
- A **TurnOffEngine** command while the **Engine** is **OFF** will damage the **Engine.**

Finally, the **Engine** is equipped with internal safety checks performed before starting up. Depending on the results of these checks, the **Engine** might fail to start.

[1] *https://www.verum.com/*
[2] *https://github.com/dezyne/community*

A controller for the simple **Engine** shall ensure these restrictions. To assist operators, the controller shall raise exceptions to a dedicated exception handler component. Typically, such a component will forward exceptions to a user interface.

2.2 Dezyne

Dezyne is a language and a set of associated tools that enable software Engineers to create, explore, and formally verify component based designs for embedded and technical software systems. A Dezyne system consists in components communicating via their exposed interfaces. Each interface is composed of a signature and a behaviour. The signature describes the possible in- and out-events while the behaviour specifies the possible sequences over these events. A unique feature of Dezyne is formal verification. The verifier proves that each component implements the behaviour specified at its interfaces. The result is a system that is proven to behave according to the specified interfaces. Code is generated from verified components. The code generator guarantees the semantics equivalence between the generated code and the verified models.

2.3 The State Pattern

We consider an object (called **Context)** that contains state driven behaviour. The main objective of the state pattern is to provide programmers with an implementation for state machines that adheres to the open/closed principle. In the pattern, the state behaviour can be modified or extended without affecting the essential parts of the **Context** object. Figure 1 shows a possible class diagram describing the state pattern.

The **Context** object receives commands from its environment, say, **cmdA** and **cmdB.** These commands trigger the execution of some actions provided by interface **IActions.** The selection of the action depends on the current state of the object. The **Context** delegates the management of states and transitions to interface **IAbstractState.** The actual states in which the **Context** can be are concrete realisations of this interface. The single responsibility of these concrete states is to handle transitions, that is, to decide the actions to be executed given the current command and state, and to determine the next state. The execution of the actions is delegated to the **IActions** interface.

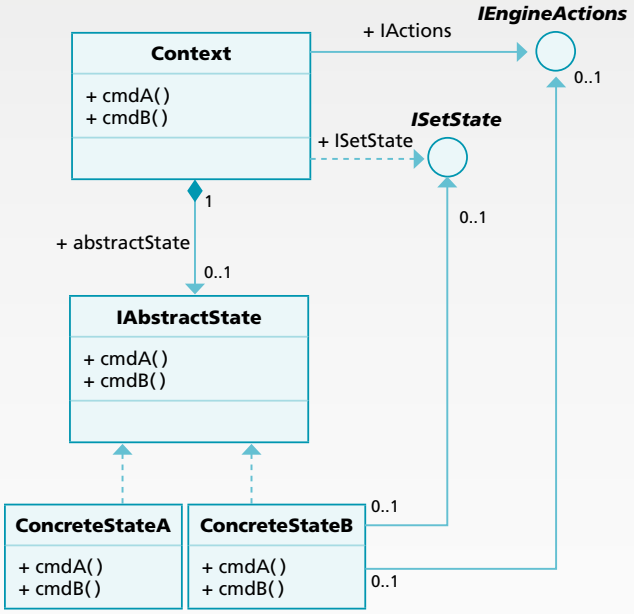


Figure 1: class diagram for the state pattern

The interaction between the different classes is illustrated in the sequence diagram in Figure 2. Assume that initially the **Context** is in state **ConcreteStateA**. When the **Context** receives command **cmdA**, it forwards it to the current concrete state object, actually via interface **IAbstractState**. The state object decides to execute **actionA()**. It then creates concrete object **ConcreteStateB** and tells the **Context** to set its current state to **ConcreteStateB**. When the next command is received, the selection of the action to be executed and the computation of the next state are delegated to concrete state **ConcreteStateB.**

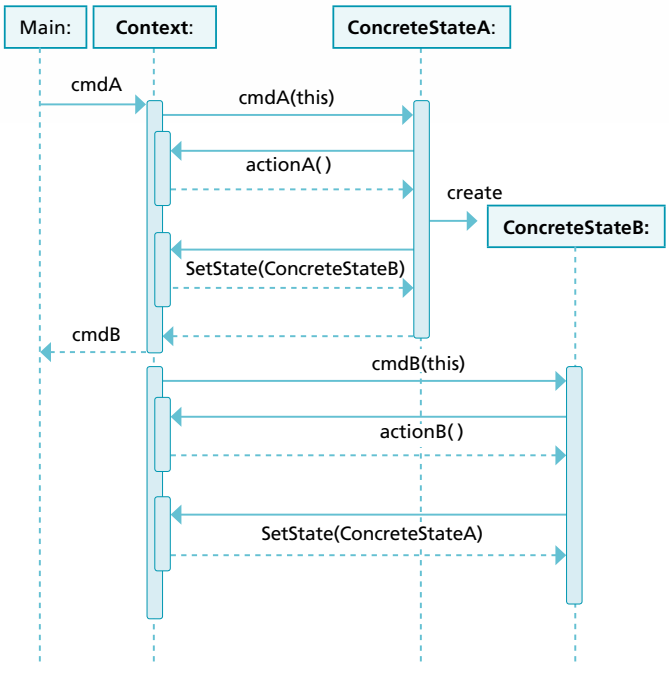


Figure 2: sequence diagram illustrating the state pattern

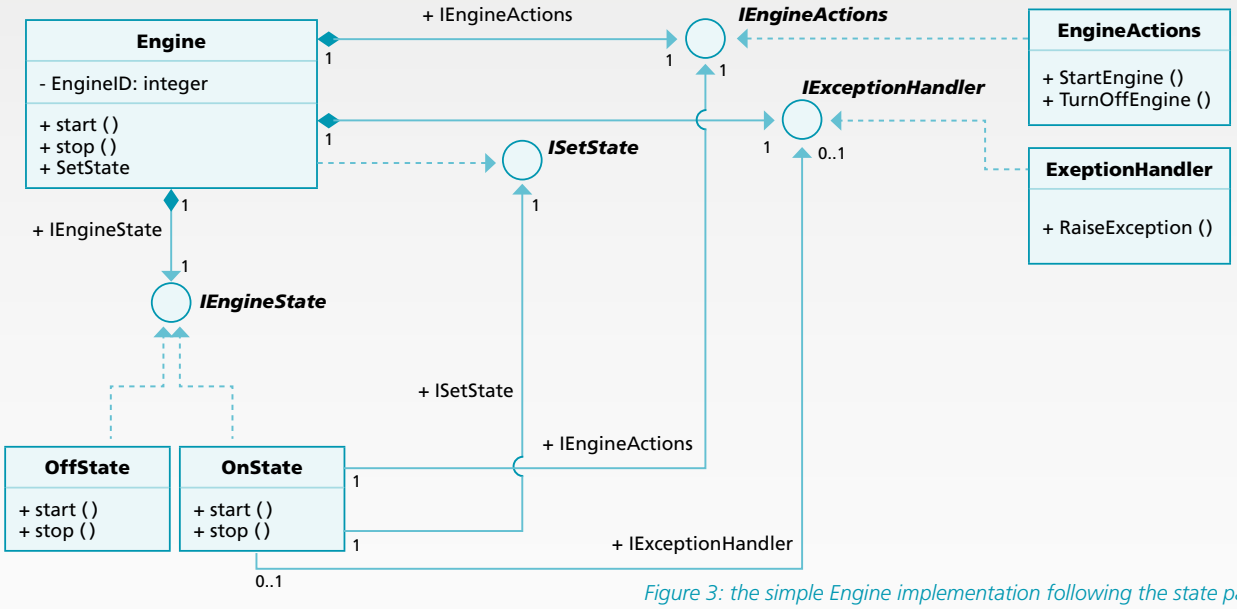


Figure 3: the simple Engine implementation following the state pattern

3 IMPLEMENTATION OF THE SIMPLE Engine USING THE STATE PATTERN

Figure 3 shows a class diagram of our implementation of the simple **Engine** following the state pattern. The **Context** object is an **Engine** object. The **Engine** inherits from the **ISetState** interface and implements the **SetState** method that effectively changes the current state. The **Engine** has pointers to interfaces **IEngineActions** and **IExceptionHandler** realized by the **EngineActions** and **ExceptionHandler** classes. These interfaces provide the **Engine** with methods to control the actual **Engine** and to raise exception. The computation of the state transitions is delegated to the **IEngineState** interface. Two concrete states may be created: state **OffState** and state **OnState.**

Figure 4 shows a sequence diagram which depicts the object interaction behaviour of the state pattern for the **Engine**. When a start is received, the **Engine** forwards it to the current state, stored in private variable **m_state:**

```
bool Engine::start()
{
  std::cout << m_intEngineID << " Engine
-> " << "\n";
  bool res = m_state->start();
  return res;
}
```

The object **OffState** then triggers the execution of the **StartEngine** method of **EngineActions**. If the start is successful, a new state is created and the **Engine** (that is, the **Context** object) current state is updated by calling method **SetState**. If the **Engine** failed to start, only an exception is raised. Below we show the code handling the start command from state **OffState**. Variables **m_IEngineActions**, **m_ISetState**, and **m_IExceptionHandler** are references to the suggested interfaces.

```
bool OffState::start()
{
  bool res = false;
  res = m_IEngineActions.StartEngine();
  if (res) {
      std::shared_ptr<IEngineState>
  newState(new OnState(m_ISetState,
                    m_IEngineActions,m_
  IExceptionHandler));
      m_ISetState.SetState(newState);
  }
  else {
      m_IExceptionHandler.
  RaiseException("Start failed !");
  }
  return res;
}
```

The transition from state **OnState** back to state **OffState** is handled in a similar way. In the next section, we show how to replace the parts of the code handling states and transitions with code generated from a Dezyne model.

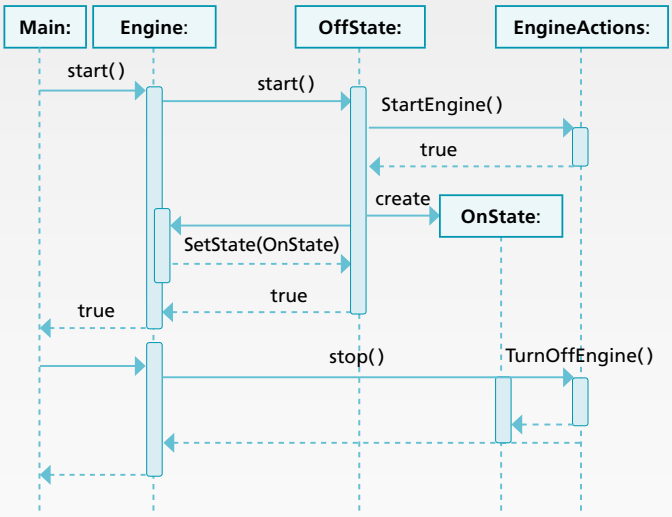


Figure 4: a sequence diagram illustrating the transition starting up the Engine

4 INTRODUCING MODEL DRIVEN TECHNOLOGIES: DEZYNE FOR STATE BEHAVIOUR

Our systematic approach to introduce Dezyne code proceeds in the following steps:

1. Create the specification of the core state machine
2. Model the required external interfaces and components
3. Create the robust component, called the armour component
4. Implement the core component
5. Create the Dezyne system
6. Generate and integrate code

Step 1: Specification of the core state machine

The objective of this step is to create the interface specification for the core state machine, that is, the code handling states and transitions. The specification of the interface describes the behaviour seen by clients of that interface. This visible behaviour is easily extracted by looking at only the states and transitions in the original code. The code for method **OffState::start()** described earlier shows that after a **start** command a possible transition from state **OFF** to state **ON** occurs or the **Context** remains in the current state. Figure 5 shows the Dezyne specification of this visible behaviour. The interface can react on the commands **start** and **stop**. The interface initially starts in state **OFF.** In that state, a **start** command can either result in a transition to state **ON** with a return result **true** or result in a self-transition (no state change) with a return result **false**. In that state a command **stop** is illegal. In state **ON**, the only legal action is **stop** and brings the machine back to state **OFF.** The right part of Figure 5 shows the state chart generated for that interface.

```
interface IDznSimpleEngine {
    in bool start();
    in void stop();

    behaviour {
        state_t state = state_t.OFF;

        [state.OFF] {
            on start : {
                state = state_t.ON;
                reply(true);
            }
            on start : {
                reply(false);
            }
            on stop : illegal;
            }

        [state.ON] {
            on start : illegal;
            on stop : {
                state = state_t.OFF;
}}}}
```

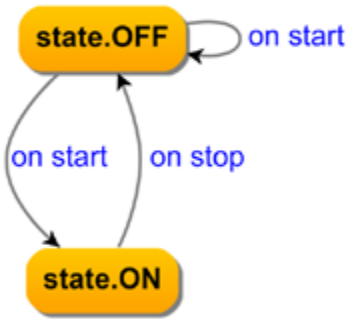


Figure 5: the core FSM and its specification

Step 2: Model the required external interfaces and components

Interfaces **IEngineActions** and **IExceptionHandler** will not be implemented in Dezyne as they do not belong to any state related. In Dezyne terminology, we call the components implementing these interfaces native. These interfaces are kept very simple. They define possible events and have no restriction about when these events are possible. For instance, the model of interface **IEngineActions** is as follows:

```
interface IDznEngineActions {
  in bool StartEngine();
  in void TurnOffEngine();

  behaviour {
       on StartEngine : reply(true);
       on StartEngine : reply(false);
       on TurnOffEngine : {}
  }
}
```

It states that **StartEngine** can be called at any time and its return value can be either **true** or **false**. It is always possible to call **TurnOffEngine**. We then create a native component for that interface, that is, a component without behaviour:

```
component DznEngineActions {
 provides IDznEngineActions
pEngineActions;
}
```

Step 3: Create the armour component

The objective of this step is to create an armour for the core state machine. This armour will only forward legal calls and will raise exceptions for illegal ones. To make the information about robustness available to environments using our robust simple **Engine**, we modify the return values for commands **start** and **stop** to the following enumeration:

```
enum callResult_t {Succeeded, Failed,
Illegal};
```

The meaning of each enumeration literal is as follows:

- Succeeded: the call was legal and completed successfully.
- Failed: the call was legal and failed to complete.
- Illegal: the call was illegal.

The signature of the robust interface is the following:

```
interface IDznSimpleEngineRobust {
  in callResult_t start();
  in callResult_t stop();
… }
```

The return values are then used to replace the illegals to proper return values. The following code shows this for state **ON** in Figure 5. The illegal is replaced with a return value **callResult_t.Illegal:**

```
[state.ON] {
  on start : {
       reply(callResult_t.Illegal);}
  on stop : {
       state = state_t.OFF;
       reply(callResult_t.Succeeded);}}
```

The robust component provides this robust interface and requires the core interface and the interface to the exception handler. The signature of the component is the following:

```
component DznSimpleEngineArmour {
  provides IDznSimpleEngineRobust
pSimpleEngineRobust;
  requires IDznSimpleEngine rSimpleEngine;
  requires injected IDznExceptionHandler
iExceptionHandler;

  behaviour {
…}}
```

Below we show the behaviour in state **OFF** showing the use of the **callResult_t** enumeration:

```
[state.OFF] {
  on pSimpleEngineRobust.start() : {
       bool res = rSimpleEngine.start();
       if (res) {
              state = state_t.ON;
              reply(callResult_t.
Succeeded);
       } else {
              iExceptionHandler.
RaiseException($"Start failed !"$);
              reply(callResult_t.Failed);}}
  on pSimpleEngineRobust.stop() : {
       iExceptionHandler.RaiseException(
$"Illegal stop, start Engine first !"$);
       reply(callResult_t.Illegal);}}
```

The verifier will prove that this component implements the provided interface while respecting the specifications of the required interfaces. In particular, the verifier will detect any possible illegal events forwarded to interface **IDznSimpleEngine.**

Step 4: Implement the core component

The objective of this step to create component **DznEngineFSM** implementing interface **IDznSimpleEngine** (see Figure 5). Because component **DznEngineFSM** is restricted to the interaction with the **Engine** actions and it is guarded by an armour, its implementation only has to deal with expected commands. It can ignore exceptions and other unexpected calls.

```
component DznEngineFSM {
 provides IDznSimpleEngine pSimpleEngine;
 requires IDznEngineActions rEngineActions;

 behaviour {
  state_t state = state_t.OFF;

       [state.OFF] {
       on pSimpleEngine.start() : {
              bool res = rEngineActions.
              StartEngine();
              if (res) state = state_t.ON;
              reply(res);
       }
  }
       [state.ON] {
       on pSimpleEngine.stop() : {
              rEngineActions.TurnOffEngine();
              state = state_t.OFF;
              }
  }
}
```

Notice the relation between the code written in Dezyne and the original C++ code for method **OffState::start().** The Dezyne code only contains the essential aspects, namely, the control of the sequence of actions and the computation of the state transition. There is no need to handle pointers and no need to handle exceptions as well. The former is dealt with in the Dezyne code generator. The latter is dealt with in the armour component.

Step 5: Create the Dezyne system

This step combines the previous components into a system that will later be integrated with the original C++ code. Figure 6 shows the component diagram of this system. The top level interface is the robust one. The verifier guarantees that the visible behaviour of that entire system refines the behaviour specified for the robust interface. The code for the system is as follows:

```
component DznSimpleEngineSystem {
 provides IDznSimpleEngineRobust
 pSimpleEngineRobust;
 system {
       DznSimpleEngineArmour EngineArmour;
       DznEngineFSM EngineFSM;
       DznExceptionHandler exceptionHandler;
       DznEngineActions EngineActions;
       EngineArmour.pSimpleEngineRobust <=>
       pSimpleEngineRobust;
       exceptionHandler.pExceptionHandler
       <=> *;
       EngineFSM.pSimpleEngine <=>
       EngineArmour.rSimpleEngine;
       EngineFSM.rEngineActions <=>
       EngineActions.pEngineActions;}}
```

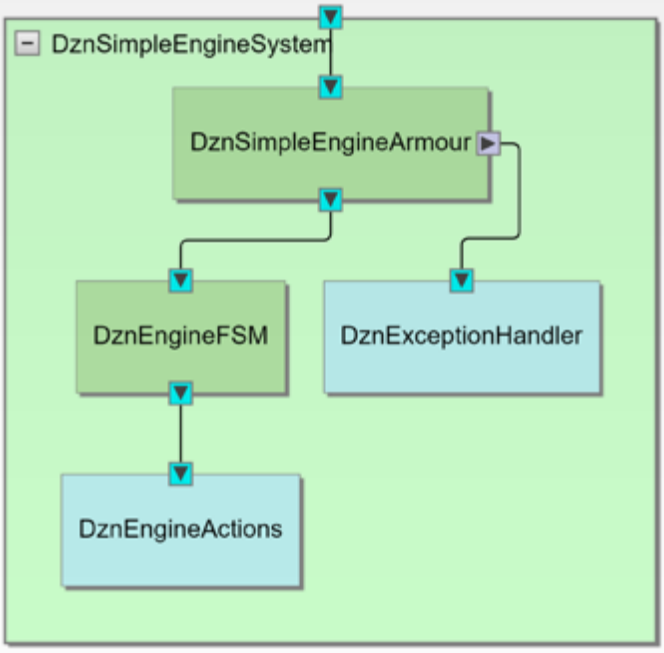


Figure 6: Dezyne component diagram

Step 6: Generate and integrate

From the verified Dezyne model, we automatically generate code. To integrate this generated code into our C++ **Engine** we need (1) to tell the Dezyne system where to find the code for the native components and (2) to tell the **Engine** where to find the Dezyne system.

For native components, Dezyne generates pure virtual classes. To link these classes to actual code, one needs to sub-class such classes. For instance, we link the **DznEngineActions** to the **IEngineActions** as follows:

```
class DznEngineActions : public
skel::DznEngineActions {
public:
  DznEngineActions(const dzn::locator&
loc) :
    m_EngineActions(loc.
get<IEngineActions>())
    , skel::DznEngineActions(loc){}

  bool pEngineActions_StartEngine()
{return m_EngineActions.StartEngine();}
  void pEngineActions_TurnOffEngine()
{return m_EngineActions.TurnOffEngine();}
private:
  IEngineActions& m_EngineActions;
};
```

The "locator" **(loc)** allows us to retrieve a reference to the interface **IEngineActions.** When constructing the **Engine** object, the locator is given a reference to that interface; actually the realisation thereof. Here is the code extract:

```
Engine::Engine(const int intEngineID) :
  m_intEngineID(intEngineID),
  m_EngineActions(new EngineActions()),
  m_exceptionHandler(new
ExceptionHandler()),
  m_DnzEngineSystem(loc.set(rt).set(*m_
EngineActions).set(*m_exceptionHandler))
{
}
```

The link between the **Engine** object and the Dezyne generated code is simply realised by calling the necessary methods of the Dezyne system. For instance, for the **stop** command:

```
void Engine::stop()
{
  std::cout << m_intEngineID << " Engine
-> " << "\n";
  m_DnzEngineSystem.pSimpleEngineRobust.
in.stop();
}
```

5 CONCLUDING REMARKS

We presented a systematic method to introduce a model-driven technology – namely, Dezyne from Verum Software Tools – in components built with the state pattern. The size of the model together with the code gluing the native components is about the same as the original C++ code. From our experience, the effort to introduce such Dezyne models and integrate the generated code is low because the state pattern already separated state transition aspects from actual actions. Still, the introduction of generated code and models already results in many benefits:

- *Synchronization between code and documentation:* The Dezyne tool set supports the generation of state charts, sequence diagrams, and component diagrams. These artefacts are automatically generated from the Dezyne textual models. In practice, designers

will create UML diagrams while creating the overall design. When implementing these design documents, code and documentation inevitably drift apart from each other as the effort to keep the code and the documentation increases significantly over time. Having the documentation produced from the code saves this effort and ensures that the code and its documentation remains synchronised.

- *Automatic creation of images:*
  From our experience, reviewing state charts is more effective than reviewing textual code. Using images, unexpected transitions, unreachable states – among others – become clearly visible. Non-specialist can look at the state machines and already ask questions about specific transitions or states. In code – even in the state pattern – states and transitions are hidden in software constructions like pointers, interfaces, etc. The benefit of using Dezyne is here that state charts, component diagrams, and sequence diagrams can be automatically generated from the model.
- *Easy code migration:*
  Because the code is generated from the model, migrating to other languages or to new version of languages amounts to push the generate code button and to re-write the glue code in the native components.
- *Big step forward:*
  Now that individual state machines are modelled in Dezyne, one can start taking further steps. For instance, one can start splitting large and complex state machines into smaller ones. This improvement can be done using the verifier as a safety net: the verifier will prove that the new structure with several state machines correctly implement the original interface. Another possible step is to start modelling cooperating state machines. Creating correct systems composed of communicating state machines is very difficult. This is where the benefits of using formal verification are the highest.

The presented method has been applied to several components in a reverse Engineering project at one of our customers. At ICT, we are convinced that model-driven technologies are means to support our customers in keeping their edge over the competition. We provide them with guidance through this fundamental transformation of the way we create software. Looking at the shortage in software designers and the speed of evolution of technologies and markets, it is time to take steps towards the digitalisation of software production.

For more information, please contact
Julien.Schmaltz@ict.nl.