C++ State Template Class Library STTCL Concept

Version 2.0 / Jan 14, 2012

Author: Günther Makulik (g-makulik@t-online.de)

Table of Contents

1 Overview

The C++ **St**ate **T**emplate **C**lass **L**ibrary provides a set of platform independent C++ template classes, that help to implement finite state machines as they are modeled with UML 2.x *State Machine Diagrams*. The template classes, their attributes and operations provide a certain mapping to the UML notation elements.

The basic approach is based on the GoF ***State*** design pattern. The template classes are designed as base of implementation classes, that mainly concentrate on the problem domain specific functionality.

1.1 The GoF State pattern

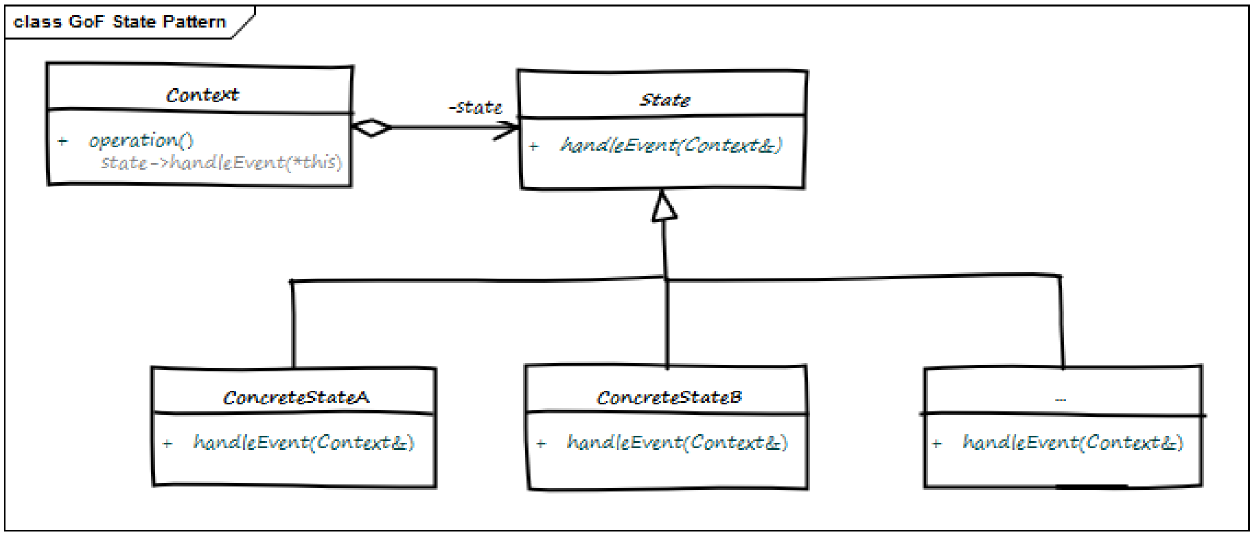


Fig 1: The structural UML representation of the GoF ***State*** *design pattern*

The class diagram in *Fig 1* shows that the **`Context`** class exposes **`operation()`** to clients and internally calls one or more **`handleEvent()`** operations of the actual **`state`** member. The **`State`** class is abstract, and the state specific behavior is implemented in the **`ConcreteStateA`**, **`ConcreteStateB`**, ... classes. Changing the **`state`** reference member will change the behavior of a **`Context`** class object, as it would have been replaced with another class.

Fig 2 Shows a refined *UML Class Diagram* of the GoF ***State*** design pattern to depict a proposal for some implementation details.

The **`Context`** class provides a method **`changeState()`** that is called by **`State`** class instances to change to a new (sibling) **`State`** instance (**`State`** is a friend class to **`Context`** to enable this). The **`changeState()`** method calls some standard state actions defined according UML state diagram notations in the right order:

1. endDo() of the current state
2. exit() of the current state
3. enter() of the new state
4. startDo() of the new state

Fig 3 Shows a simple UML State Machine Diagram, that uses some basic UML State Machine Diagram notation elements.

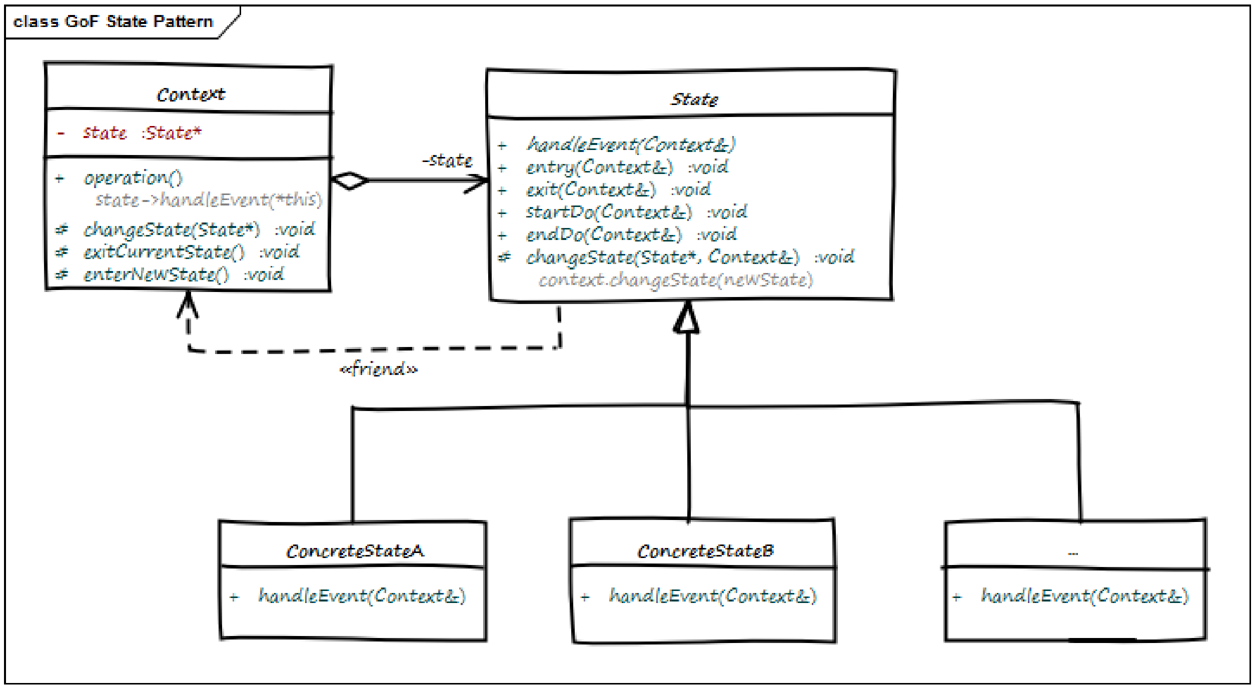


Fig 2: GoF State design patten implementation details

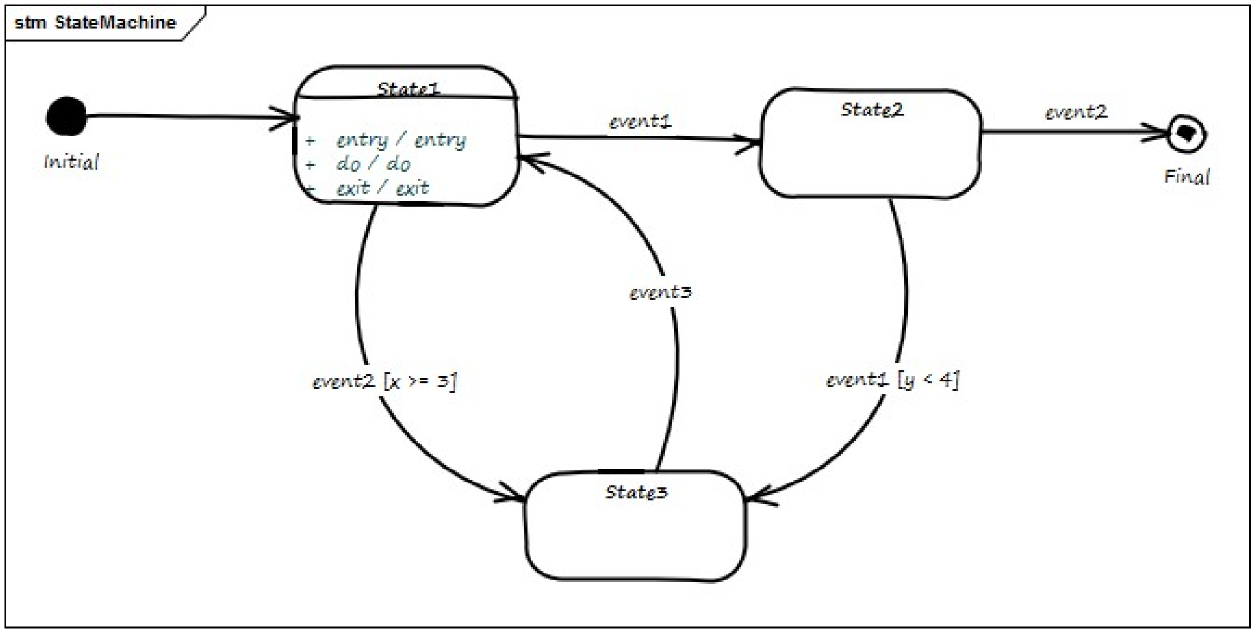


Fig 3: Simple sample UML State Machine Diagram

The state machine starts from *Initial* with a direct transition to *State1*. *State1* has *entry()*, *do()* and *exit()* actions defined. If *event1* is triggering the state machine *State1* transits to *State2*, if *event2* is triggering the state machine and the context variable *x* is greater or equal than 3 *State1* transits to *State3*. Aso from the other states.

How does this map to the refined GoF ***State*** design pattern model? The following diagrams illustrate how this would be implemented.

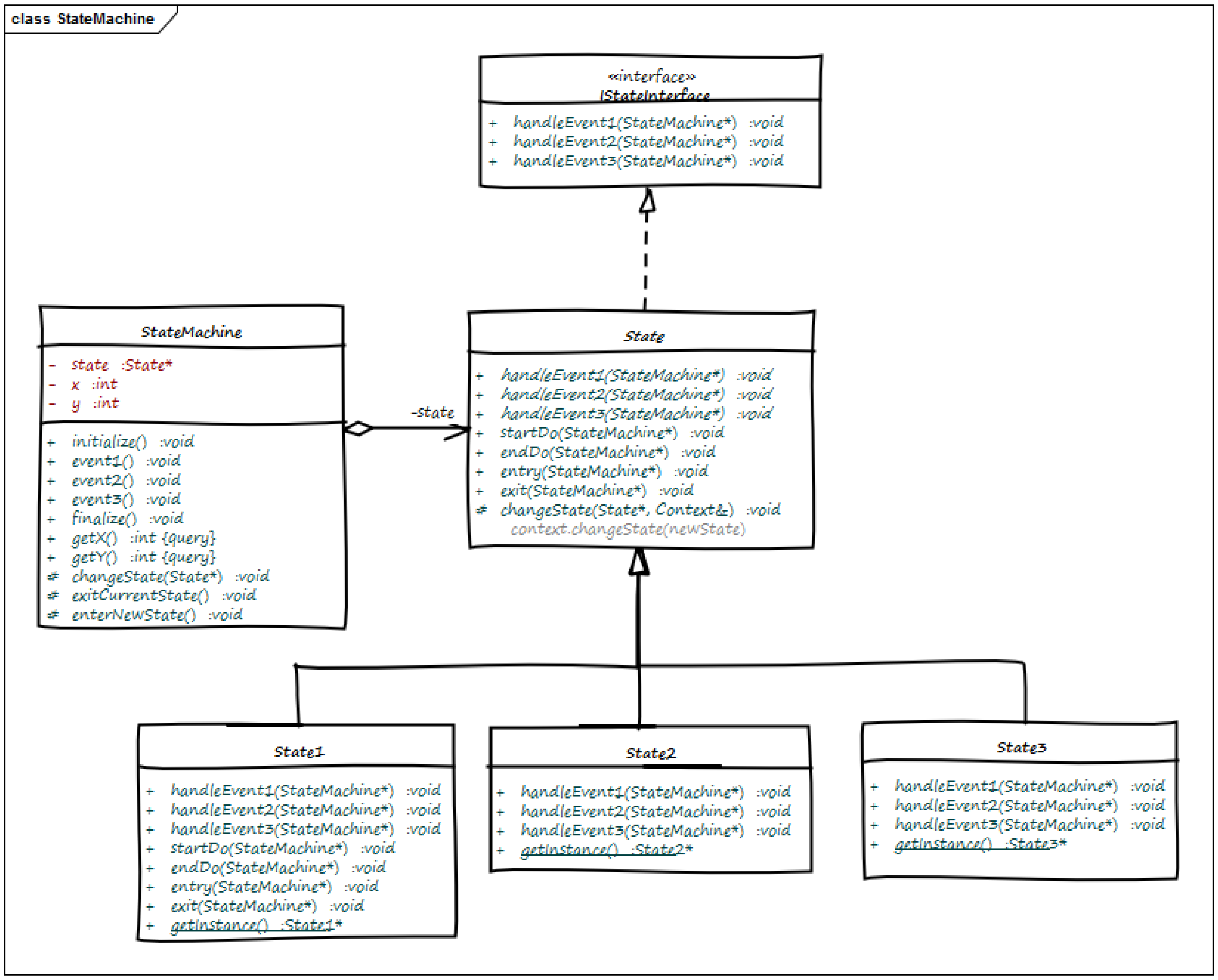

class StateMachine
«interface»
IStateInterface
+ handleEvent1(StateMachine*) :void
+ handleEvent2(StateMachine*) :void
+ handleEvent3(StateMachine*) :void
StateMachine
- state :State*
- x :int
- y :int
+ initialize() :void
+ event1() :void
+ event2() :void
+ event3() :void
+ finalize() :void
+ getX() :int {query}
+ getY() :int {query}
# changeState(State*) :void
# exitCurrentState() :void
# enterNewState() :void
-state
State
+ handleEvent1(StateMachine*) :void
+ handleEvent2(StateMachine*) :void
+ handleEvent3(StateMachine*) :void
+ startDo(StateMachine*) :void
+ endDo(StateMachine*) :void
+ entry(StateMachine*) :void
+ exit(StateMachine*) :void
# changeState(State*, Context&) :void
context.changeState(newState)
State1
+ handleEvent1(StateMachine*) :void
+ handleEvent2(StateMachine*) :void
+ handleEvent3(StateMachine*) :void
+ startDo(StateMachine*) :void
+ endDo(StateMachine*) :void
+ entry(StateMachine*) :void
+ exit(StateMachine*) :void
+ getInstance() :State1*
State2
+ handleEvent1(StateMachine*) :void
+ handleEvent2(StateMachine*) :void
+ handleEvent3(StateMachine*) :void
+ getInstance() :State2*
State3
+ handleEvent1(StateMachine*) :void
+ handleEvent2(StateMachine*) :void
+ handleEvent3(StateMachine*) :void
+ getInstance() :State3*

Fig 4: Class diagram for StateMachine

The **`StateMachine`** class provides methods to trigger the state machines events (**`event1()`**, **`event2()`** and **`event3()`**), additionally the **`initialize()`** and **`finalize()`** methods are provided to represent the transitions from the *Initial* and to the *Final* pseudo states.
The event triggering methods delegate the behavior to the current state of the state machine using the **`IStateInterface`**. For that particular situation the inherited **`State`** classes don't maintain any internal state (variables), and are imlemented as singletons. The **`getInstance()`** methods return the singleton instances.
The following sequence diagrams exemplary show behavior when **`StateMachine`** methods are called by a client class.

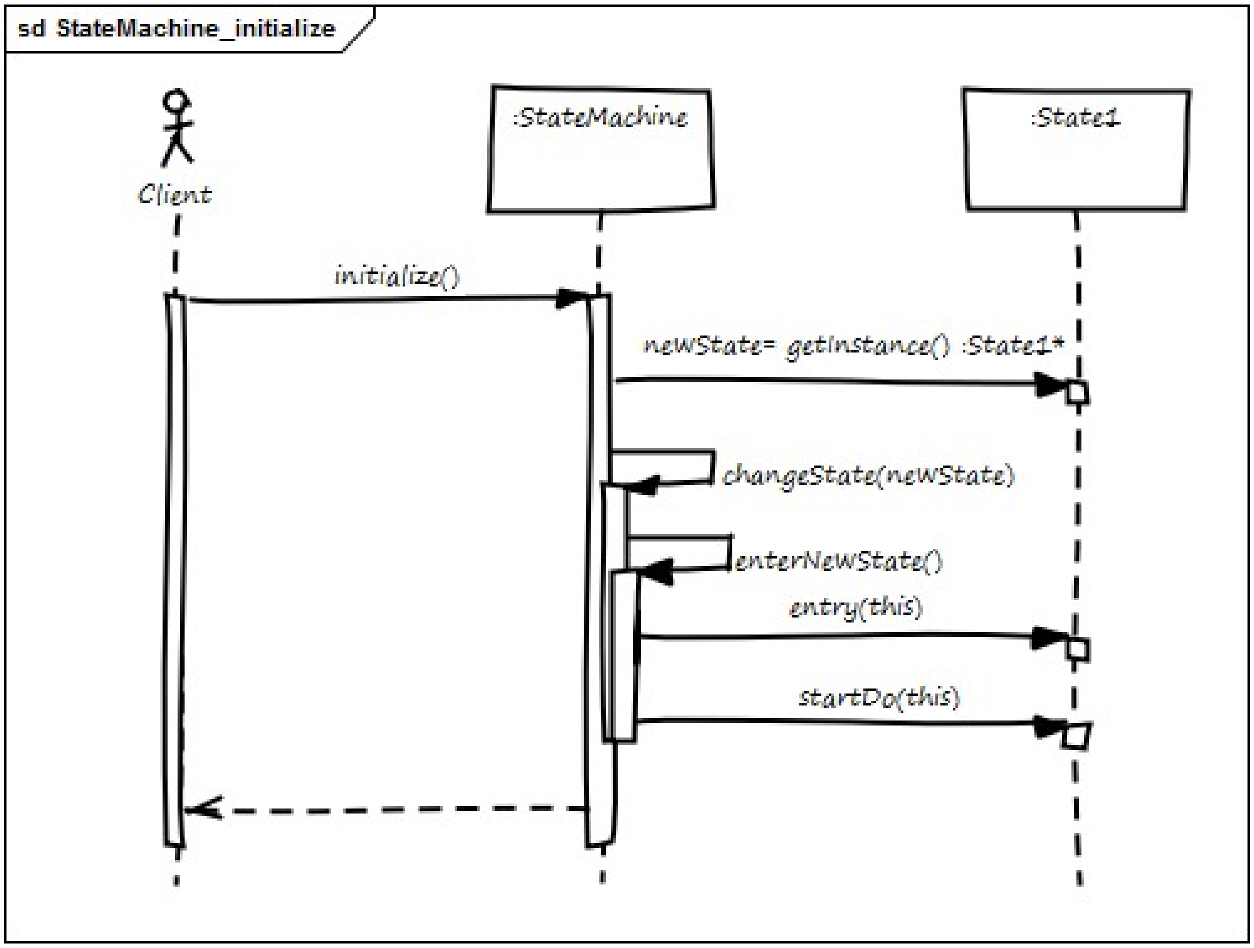


Fig 5: Initializing the StateMachine

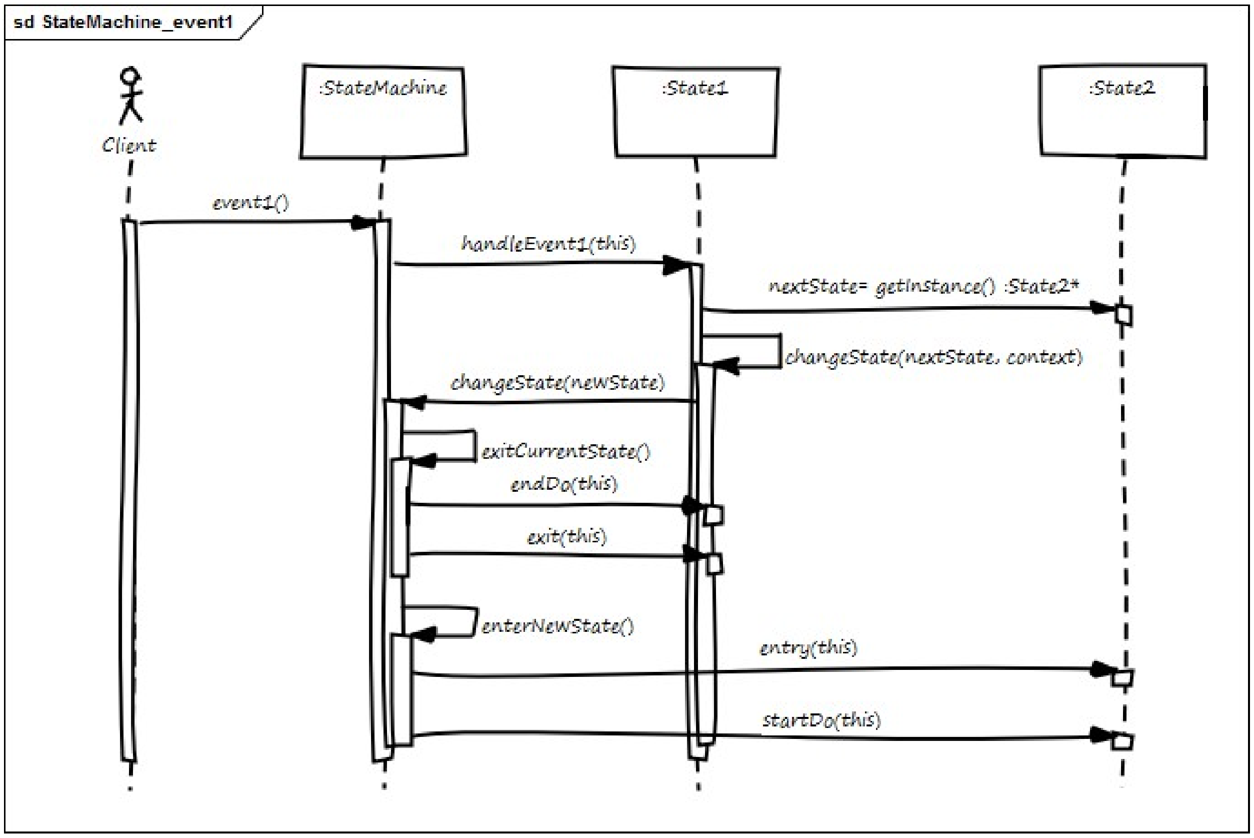


Fig 6: Call of event1() when current state is State1

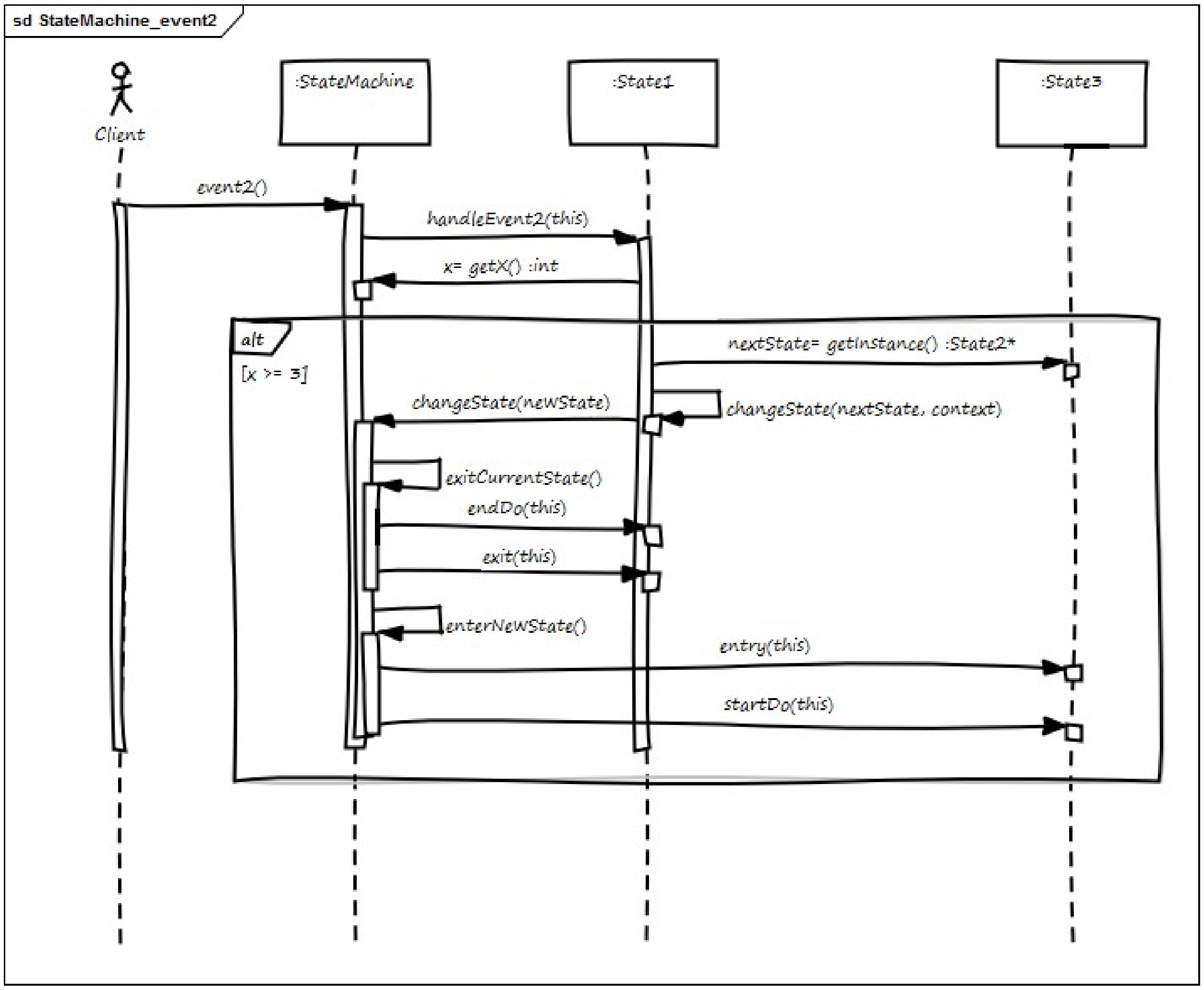
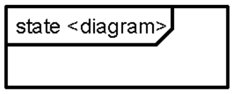
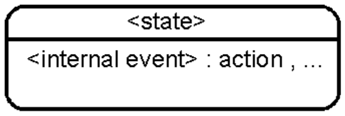
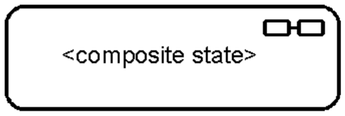
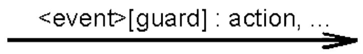


Fig 7: Call of event2() when current state is State1

1.2 *General mapping of UML State Machine notation elements*

The following table shows how the basic *UML State Machine Diagrams* notation elements map to the elements described in the GoF ***State*** design pattern:

UML state machine diagram notation	GoF *State* design pattern element
 The state machine diagram itself	A **`Context`** class instance.
 A state (atomic)	A ***`State`*** class instance (**`ConcreteStateA`**, **`ConcreteStateB`**).
<Internal event>:= enter, do, exit, <event> A state internal event. The do event is intrinsically triggered by the internal enter event.	A call to the ***`State`*** classes **`entry()`**, **`exit()`** or **`do()`** operation.
 A composite state. The internal states are modeled in a sub-state machine diagram	A ***`State`*** class instance, that also serves as another **`Context`** classes instance.
 A transition between two states	A call of the **`Context::changeState()`** operation.
<event> An event that triggers the associated transition	A call to a public **`Context::operation()`** operation, that delegates behavior to a ***`State::handleEvent()`*** operation. All events visible in the state machine diagram can be triggered via the public **`Context::operation()`** operations.
[guard] A conditional expression, that must return true to execute the associated transition. Guard conditions that are associated to the same source state must be mutually exclusive.	A conditional statement inside the **`Context::operation()`** or ***`State::handle()`*** operations, that decides to call **`Context::changeState()`**.
: action, ... A list of specified event triggered operations	The specified action operations are called inside the implementation ***`State::handleEvent()`*** event handler operation[1]. The calling order may be unspecified.

[1] action operations that appear on a transition are not allowed to access the contexts state. In fact these operations

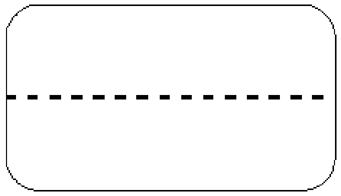
Table 1: Basic UML State Machine Elements

should be performed after the current state was exited and before the new state is entered. That's difficult to achieve with the GoF State design pattern, since changing state is an atomic operation in the Context class.

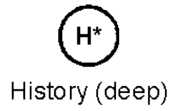
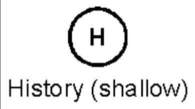
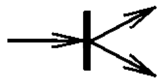
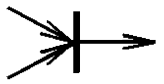
There's a number of advanced *UML State Machine Diagram* notation elements, that do not directly map to any conceptual element described in the GoF ***State*** design pattern. These can be mapped to certain aspects of implementation and behavior though.

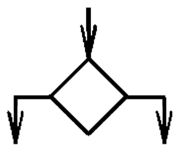
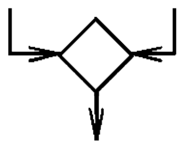
This mainly concerns the so called pseudo-states, that also have incoming and/or outgoing transitions. But vs. concrete states, pseudo-states only a kind of transient states, that represent complex or intrinsic transition paths in a state machine diagram or composite state.

The following table lists possible implementation approaches for further *UML State Machine Diagram* notation elements:

UML state machine diagram notation		**GoF *State* design pattern implementation**
	Separates concurrently active regions within a composite state or state machine diagram.	This requires concurrent program execution mechanisms (e.g. threading) supported by an operating system. Each region defines an associated ***`State`*** class reference, to delegate the event handling to a ***`State::handleEvent()`*** operation concurrently. A single region can also be considered as a concrete composite state implementation, that supports a non blocking ***`State::do()`*** operation, that executes asynchronously in a loop. Further it is necessary to have a mechanism to propagate events to the asynchronously executed operation loop.

UML state machine diagram notation	**GoF *State* design pattern implementation**
Pseudo-States	
● Initial — An entry point of the state machine diagram. Initial pseudo-states never have the target role of a transition.	A constructor call to create a `Context` class instance in the simplest case. The constructor calls the `Context::changeState()` operation to set the initial *`State`* reference. If any events and/or guards are specified for the associated outgoing transitions, the `Context` class should provide a property attribute to check it's initialization status, and leave the decision, which initial *`State`* reference to set, to the associated `Context::operation()` event operations. This behavior may be encapsulated in a `Context::initialize()` operation.
◉ Final — An exit point of the state machine diagram. Final pseudo-states never have the source role of a transition.	A *`State`* class implementation, that never calls the `Context::changeState()` operation.
⊗ Exit — An exit point of a state machine or composite state triggered by the incoming transition's event.	A `Context::finalize()` operation, that calls the actual *`State`* references `exit()` operation.
✕ Terminate — Exits the composite state or state machine triggered by the incoming transition's event.	A destructor call to a `Context` class instance in the simplest case.

UML state machine diagram notation	GoF *State* design pattern implementation
 Represents the most recent active configuration of a composite state. In opposite to the shallow history pseudo-state, this includes all sub states of all regions and their recently active sub states recursively.	The composite **`Context`** class instance must keep track of the most recent sub ***`State`*** reference, when the composite ***`State`*** classes **`exit()`** operation is called. When the composite ***`State`*** classes **`enter()`** operation is called later on, the composite **`Context`** class directly transits to the remembered most recent sub ***`State`*** reference.
 Represents the most recent sub state of a composite state. A composite state can have at most one history pseudo-state. At most one transition to the default sub state may originate the history pseudo-state. This transition is executed in case the composite state was never active before.	The same behavior as described for the deep History. But In case, that a reentered sub ***`State`*** reference also represents a composite state, it's composite **`Context`** class must be (re-)initialized.
 A Fork. Serves to split a single incoming transition into concurrently executed outgoing transitions. No guards are allowed on any associated transitions.	A fork represents the initiation of concurrently executed operations (i.e. tasks, threads) of a composite **`Context`** class. This can be implemented as a non blocking operation that starts all of the associated concurrently executed operations.
 A Join. Serves to synchronize multiple concurrently executed incoming transitions into a single outgoing transitions. No guards are allowed on any associated transitions.	A join represents a synchronization point (i.e. semaphore, mutex) for formerly initiated concurrently executed operations of a composite **`Context`** class. This can be implemented in a blocking operation, that waits on completion of all the associated concurrently executed operations.

UML state machine diagram notation	**GoF *State* design pattern implementation**
 A Choice. Serves to select the outgoing transitions according runtime conditions represented by the guards, associated to them. The guard conditions must be mutually exclusive, to choose a certain transition path. The model requires at least one of the guard conditions to evaluate to true, therefore one of the outgoing transitions should cover the else/default case. Unlike the Fork pseudo-state, a Choice node doesn't initiate any concurrently executed transitions.	A Choice can be implemented as a **`if ...else if ..else`** or **`switch`** conditional block in a ***`State`*** classes implementation (**`ConcreteStateA`**, **`ConcreteStateB`**)., that choose the appropriate target ***`State`*** reference parameter for a call to the **`Context::changeState()`** operation. **Note:** Choices don't serve to split transition paths into concurrently executed operations!
 A Merge. Serves to combine alternate execution flows into a single outgoing transition. Unlike the Join pseudo-state, a Merge node doesn't provide synchronization of concurrently executed transitions, that originate from different regions. Also a Merges incoming transitions may have guard conditions associated.	A Merge can be implemented as an operation, that is shared by a number of ***`State`*** class implementations (**`ConcreteStateA`**, **`ConcreteStateB`**), and ends up in a single call of **`Context::changeState()`** operation. The decision to call this operation is done in the implementation of the ***`State::handleEvent()`*** operation, according the associated guard condition.

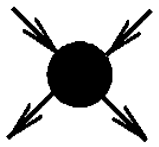
UML state machine diagram notation	GoF *State* design pattern implementation
 A Junction. Serves to share transition paths for the incoming transitions. The incoming and or outgoing transitions have guard conditions associated. Incoming transitions are shared between the source states. Outgoing transitions must have mutually exclusive guard conditions. The model requires at least one of the guard conditions to evaluate to true, therefore one of the outgoing transitions should cover the else/default case.	The Junction serves complex conditional path transitions that can be implemented in a similar way as the Choice and Merge pseudo-states. UML 2.1 specification restricts the guard conditions to be static (actively waiting for all incoming events). IMHO this can be interpreted, that the Junction node should be another implementation of the ***`State`*** class. Such implementation should not affect the **`Context`** classes attributes, but just serve to forward incoming events to outgoing transitions (i.e. **`Context::changeState()`** calls).

Table 2: Advanced UML State Machine Elements

2 Implementation Design

The basic implementation approach of the C++ **STTCL** is, to provide abstractions of the GoF ***State*** design patterns static structures. This is accomplished using parametrized base classes that serve certain functionality, associated to the formerly listed *UML State Machine Diagram* notation elements.

A general design goal is to avoid any dependency on specific OS environments or other libraries (including STL) as far as possible. Additionally STTCL should be optimized for a minimal footprint regarding RAM and ROM usage, but still best possible performance on any target environment. Dynamic memory allocation shouldn't be necessary at all.

The **`Context::operation()`** and ***`State::handleEvent()`*** operations, described in the design pattern, can be considered as a pair of compliant, application specific interfaces (source/sink).

The interface realized by the **`Context`** class is visible to any clients of the state machine implementation. The interface realized by the ***`State`*** class implementations, and those implementations themselves, shouldn't be visible to any clients of the state machine implementation.

STTCL provides various template base classes that can be used to implement state machines and concrete states of an application. Either state machine and state base classes use the CRTP pattern to bind to the implementation classes, this avoids C++ vtable instantiation and access as much as possible.

Fig 8 Shows the simplest possible implementation of a state machine.

Fig 8 Shows a class diagram, that illustrates a state machine implementation that uses the STTCL basic classes. The highlighted elements represent the sttcl classes.

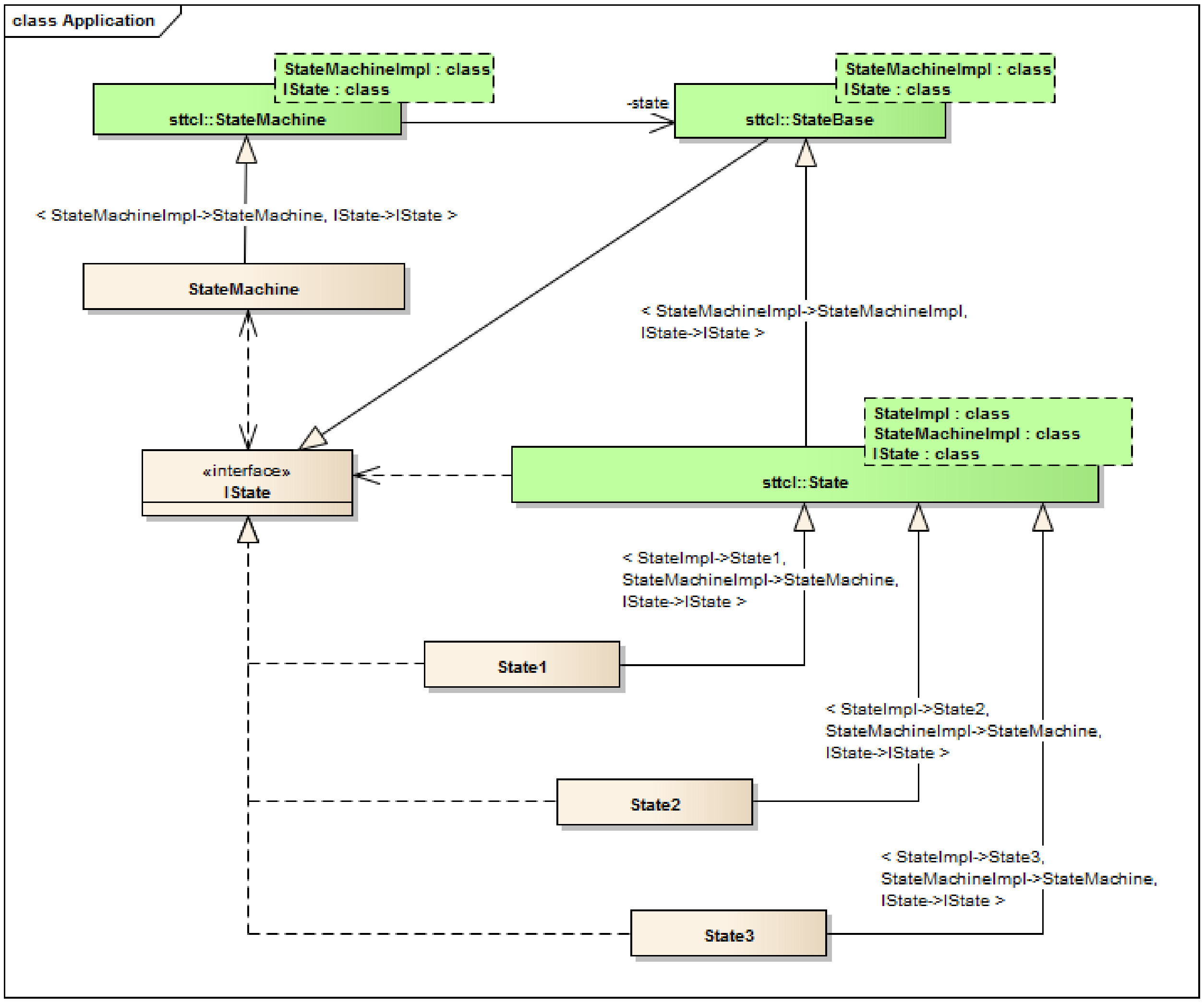


*Fig 8: The basic STTCL **State** design pattern abstraction*

The ***State*** class will provide certain common state specific operations, like **entry()**, **exit()**, **startDo() and endDo()**. These operations are not intended to be called by the implementation classes, but rather by the corresponding **StateMachine** class. The ***State*** class also provides a protected operation **changeState()**, that will delegate to a call to the ***StateMachineImpl*** context parameters **StateMachine::changeState()** operation. This operation enables the concrete state implementations, to implement transitions to another concrete state. As formerly stated, the **StateMachine::changeState()** operation shouldn't be directly accessible for any client (or actor) classes of the state machine. This requires, that the ***State::changeState()*** operation is allowed to friendly access the **StateMachine::changeState()** operation.

The **StateMachine** class mainly serves to implement the transitions' behavior, when they are enabled and passed the guard conditions implemented in a concrete state. This concerns control of the exited and entered state's synchronous and asynchronous execution behavior.

As discussed in the GoF ***State*** design pattern, the concrete state implementations may provide singleton instances (accessible though a static operation of the class), as far no state runtime attributes need to be maintained. This approach will guarantee, that the **StateMachine** doesn't need to reference concrete

`State` implementations, other than it's initial state.

2.1 Aspect Oriented Modelling of State Machines

The STTCL template classes use aspect oriented implementations to handle the various features of UML State Machine Diagrams. Aspects are injected through inherited template parameter classes. The particular aspects concern

- Composite states
  - State history behavior
- Concurrency
  - Do action handling
  - State Machine regions
  - Asynchronous event argument handling

and can be combined arbitrarily as necessary.

Since concurrency behavior is at least depend on some OS specific environment capabilities, STTCL provides some abstractions for the basic features used. These refer to

- Threads
- Mutexes
- Semaphores
- Time duration representation