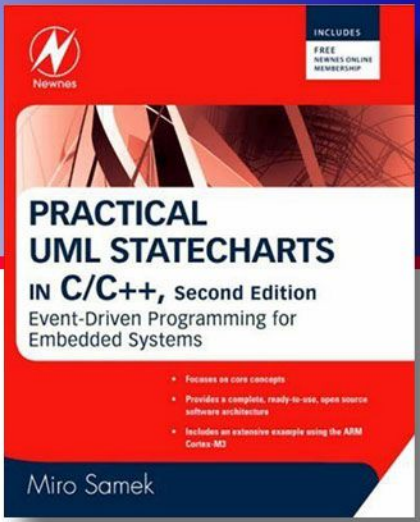
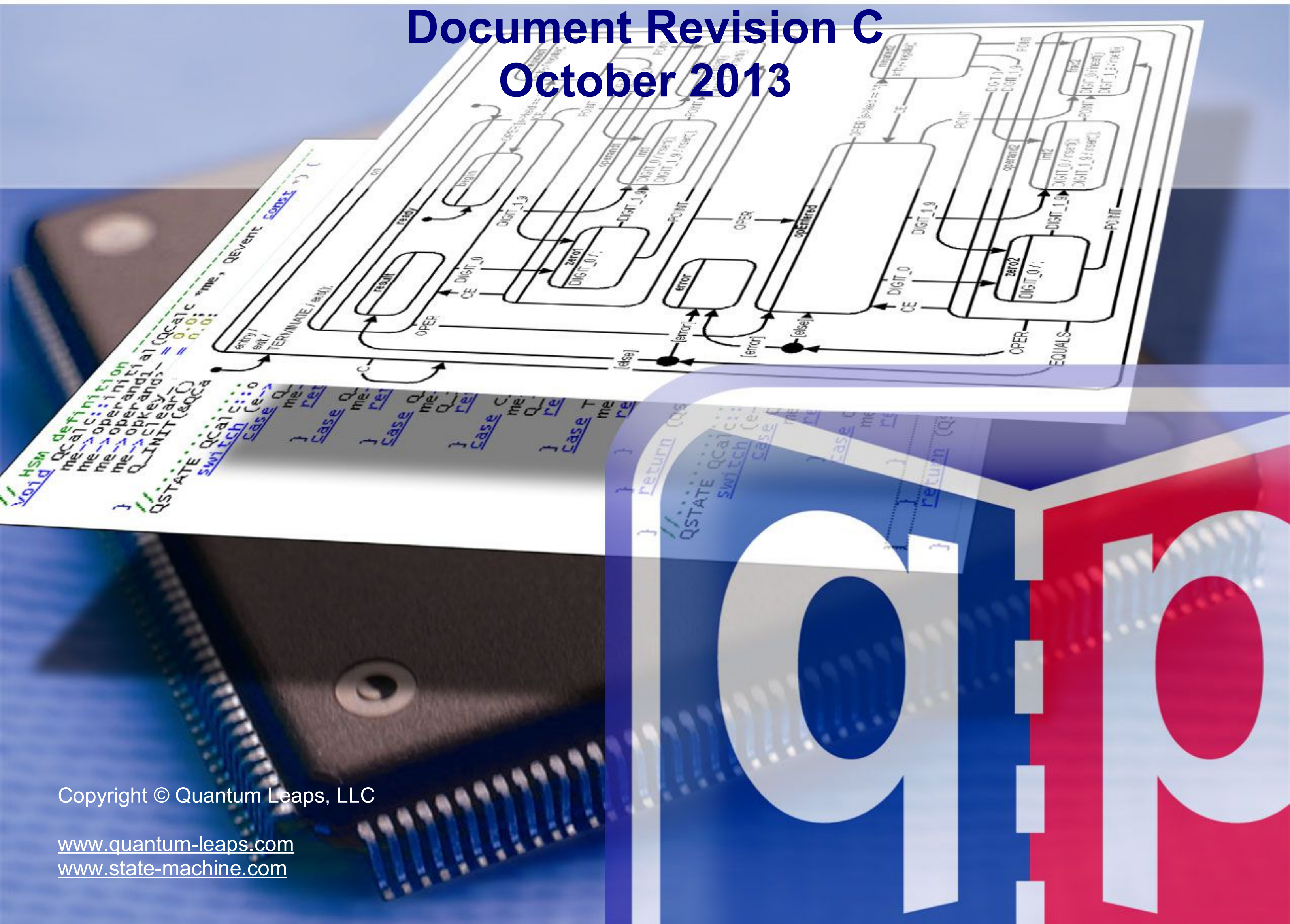
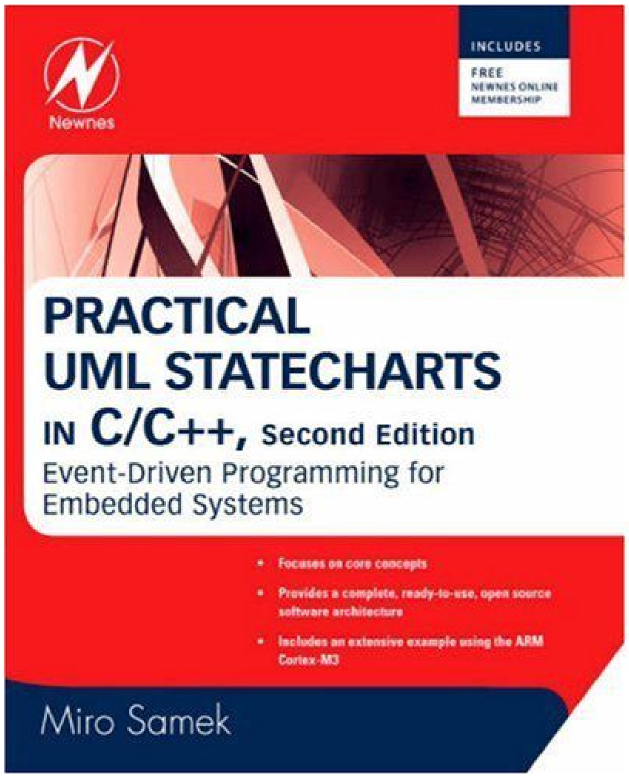
quantum™ Leaps
innovating embedded systems



Design Pattern Orthogonal Component

Document Revision C
October 2013




INCLUDES
FREE
NEWNES ONLINE
MEMBERSHIP
Newnes
PRACTICAL
UML STATECHARTS
IN C/C++, Second Edition
Event-Driven Programming for
Embedded Systems
Focuses on core concepts
Provides a complete, ready-to-use, open source software architecture
Includes an extensive example using the ARM Cortex-M3
Miro Samek

The following excerpt comes from the book *Practical UML Statecharts in C/C++, 2nd Ed: Event-Driven Programming for Embedded Systems* by Miro Samek, Newnes 2008.

ISBN-10: 0750687061
ISBN-13: 978-0750687065

Orthogonal Component

Intent

Use state machines as components.

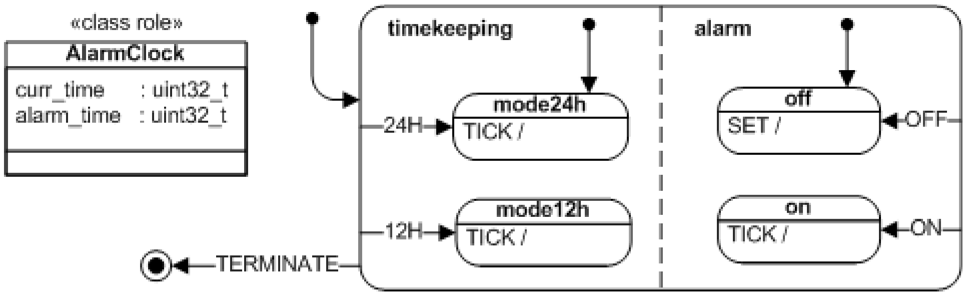
Problem

Many objects consist of relatively independent parts that have state behavior. As an example, consider a simple digital alarm-clock. The device performs two largely independent functions: a basic timekeeping function and an alarm function. Each of these functions has its own modes of operation. For example, timekeeping can be in two modes: 12-hour or 24-hour. Similarly, the alarm can be either on or off.

The standard way of modeling such behavior in UML statecharts is to place each of the loosely related functions in a separate *orthogonal region*, as shown in Figure 5.9. Orthogonal regions are a relatively expensive mechanism[1] that the current implementation of the QEP event processor does not support. Also, orthogonal regions aren't often the desired solution because they offer little opportunity for reuse. You cannot reuse the "alarm" orthogonal region easily outside the context of the `AlarmClock` state machine.

Figure 5.9 `AlarmClock` class and its UML state machine with orthogonal regions.

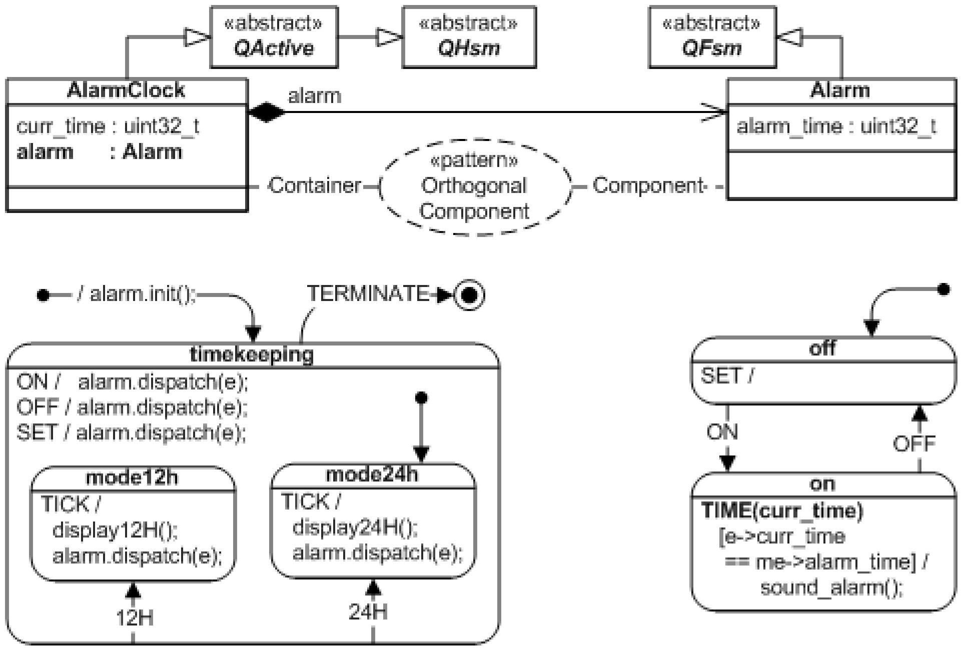
[1] Each orthogonal region requires a separate state-variable (RAM) and some extra effort in dispatching events (CPU cycles).



Solution

You can use object composition instead of orthogonal regions. As shown in Figure 5.10, the alarm function very naturally maps to the `Alarm` class that has both data (`alarm_time`) and behavior (the state machine). Indeed, Rumbaugh and colleagues [Rumbaugh+ 91] observe that this is a general rule. Concurrency virtually always arises within objects *by aggregation*; that is, multiple states of the components can contribute to a single state of the composite object.

Figure 5.10 The Orthogonal Component state pattern.



The use of aggregation in conjunction with state machines raises three questions:

1. How does the container state machine communicate with the component state machines?
2. How do the component state machines communicate with the container state machine?
3. What kind of concurrency model should be used?

The composite object interacts with its aggregate parts by synchronously dispatching events to them (by invoking `dispatch()` on behalf of the components). GUI systems, for instance, frequently use this model because it is how parent windows communicate with their child windows (e.g., dialog controls). Although, in principle, the container could invoke various functions of its components or access their data directly, dispatching events to the components should be the preferred way of communication. The components are state machines, and their behavior depends on their internal state.

You can view the event-dispatching responsibility as a liability given that errors will result if the container "forgets" to dispatch events in some states, but you can also view it as an opportunity to improve performance. Explicit event dispatching also offers more flexibility than the event dispatching of orthogonal regions because the container can choose the events it wants to dispatch to its components and even change the event type on the fly. I demonstrate this aspect, when the `AlarmClock` container generates the TIME event-on-the-fly before dispatching it to the `Alarm` components (see Listing 5.8(9)).

To communicate in the opposite direction (from a component to the container), a component needs to post events to the container. Note that a component cannot call `dispatch()` on behalf of the container because this would violate RTC semantics. As a rule, the container is always in the middle of its RTC step when a component executes. Therefore, components need to asynchronously post (queue) events to the container.

This way of communication suggests a concurrency model in which a container shares its execution thread with the state machine components[2]. The container dispatches an event to a component by synchronously calling `dispatch()` state machine operation on behalf of the component. Because this function executes in the container's thread, the container cannot proceed until `dispatch()` returns, that is, until the component finishes its RTC step. In this way, the container and components can safely share data without any concurrency hazards (data sharing is also another method of communication among them). However, sharing the container's data makes the components dependent on the container and thus makes them less reusable.

As you can see on the right-side of Figure 5.10, I decided to derive the `Alarm` component from the simpler `QFsm` base class in order to demonstrate that you have a choice of the base class for the components. You can decide to implement some components as HSMs and others as FSMs. The QEP event processor supports both options.

By implementing half of the problem (the `AlarmClock` container) as a hierarchical state machine and the other half as a classical "flat" FSM (the `Alarm` component), I can contrast the hierarchical and non-hierarchical solutions to essentially identical state machine topologies. Figure 5.10 illustrates the different approaches to representing mode switches in the HSM and in the FSM. The hierarchical solution demonstrates the "Device Mode" idiom [Douglass 99], in which the signals 12H and 24H trigger high-level transitions from the "timekeeping" superstate to the substates "mode12h" and "mode24h", respectively. The `Alarm` FSM is confined to only one level and must use direct transitions ON and OFF between its two modes. Although it is not clearly apparent with only two modes, the number of mode-switch transitions in the hierarchical technique scales up proportionally to the number of modes, *n*. The nonhierarchical solution requires many more transitions — $n \times (n - 1)$, in general — to interconnect all states. There is also a difference in behavior. In the hierarchical solution, if a system is already in "mode12h", for example, and the 12H signal arrives, the system leaves this mode and re-enters it again. (Naturally, you could prevent that by overriding the high-level 12H transition in the "mode12h" state.) In contrast, if the flat state machine of the `Alarm` class is in the "off" state, for example, then nothing happens when the OFF signal appears. While this solution might or might not be what you want, the hierarchical solution (the Device Mode idiom) offers you both options and scales much better with a growing number of modes.

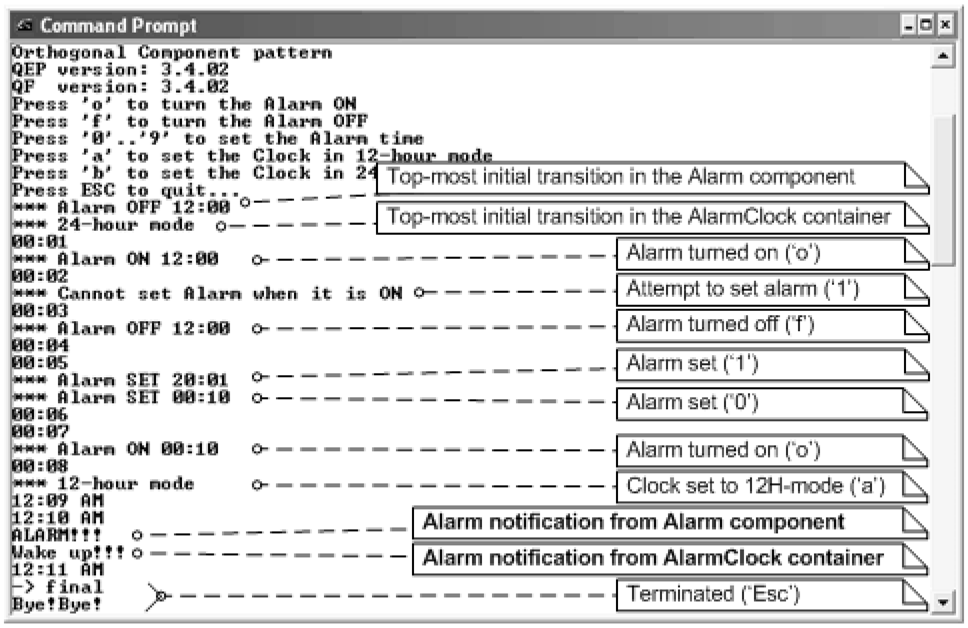
Sample Code

The sample code for the Orthogonal Component state pattern is found in the directory `qpc\examples\win32\mingw\comp\`. You can execute the application by double-clicking on the file `COMP.EXE` file in the `dbg\` subdirectory. Figure 5.11 shows the output generated by the `COMP.EXE` application. The application prints the status of every mode change, both in the `AlarmClock` container and in the `Alarm` component. Additionally, you get feedback about the currently set alarm time and a notification when the alarm time is reached. The legend of the key-strokes at the top of the screen describes how to generate events for the

[2] Most commonly, all orthogonal regions in a UML statechart also share a common execution thread [Douglass 99].

application. Also, please note that to make things happen a little faster, I made this alarm clock advance by one accelerated minute per one real second.

Figure 5.11 Annotated output generated by `COMP.EXE`.


Command Prompt
Top-most initial transition in the Alarm component
Top-most initial transition in the AlarmClock container
Alarm turned on ('o')
Attempt to set alarm ('1')
Alarm turned off ('f')
Alarm set ('1')
Alarm set ('0')
Alarm turned on ('o')
Clock set to 12H-mode ('a')
Alarm notification from Alarm component
Alarm notification from AlarmClock container
Terminated ('Esc')

```
Orthogonal Component pattern
QEP version: 3.4.02
QF  version: 3.4.02
Press 'o' to turn the Alarm ON
Press 'f' to turn the Alarm OFF
Press '0'..'9' to set the Alarm time
Press 'a' to set the Clock in 12-hour mode
Press 'b' to set the Clock in 24
Press ESC to quit...
*** Alarm OFF 12:00
*** 24-hour mode
00:01
*** Alarm ON 12:00
00:02
*** Cannot set Alarm when it is ON
00:03
*** Alarm OFF 12:00
00:04
00:05
*** Alarm SET 20:01
*** Alarm SET 00:10
00:06
00:07
*** Alarm ON 00:10
00:08
*** 12-hour mode
12:09 AM
12:10 AM
ALARM!!!
Wake up!!!
12:11 AM
-> final
Bye!Bye!
```

The sample code demonstrates the typical code organization for the Orthogonal Component state pattern, in which the component (`Alarm`) is implemented in a separate module from the container (`AlarmClock`). The modules are coupled through shared signals, events, and variables (Listing 5.5). In particular, the pointer to the container active object `APP_alarmClock` is made available to all components, so that they can post events to the `AlarmClock` container.

Listing 5.5 Common signals and events (file `clock.h`).

```
#ifndef clock_h
#define clock_h

enum AlarmClockSignals {
    TICK_SIG = Q_USER_SIG,                                 /* time tick event */
    ALARM_SET_SIG,                                           /* set the alarm */
    ALARM_ON_SIG,                                        /* turn the alarm on */
    ALARM_OFF_SIG,                                      /* turn the alarm off */
    ALARM_SIG,  /* alarm event from Alarm component to AlarmClock container */
    CLOCK_12H_SIG,                                  /* set the clock in 12H mode */
    CLOCK_24H_SIG,                                  /* set the clock in 24H mode */
    TERMINATE_SIG                                /* terminate the application */
};
/*..........................................................................*/
typedef struct SetEvtTag {
    QEvent super;                                       /* derive from QEvent */
    uint8_t digit;
} SetEvt;
```

```
    typedef struct TimeEvtTag {
        QEvent super;                                   /* derive from QEvent */
        uint32_t current_time;
    } TimeEvt;

    extern QActive *APP_alarmClock;        /* AlarmClock container active object */

    #endif                                                          /* clock_h */
```

Note: Please note that the `APP_alarmClock` pointer has the generic type `QActive*`. The components only "know" the container as a generic active object, but don't know its specific data structure or state handler functions. This technique is called *opaque pointer* and is worth remembering for reducing dependencies among modules.

Listing 5.6 shows the declaration of the `Alarm` component (see Figure 5.10). Please note that I actually don't need to expose the state-handler functions in the `alarm.h` header file. Instead, I provide only the generic interface to the `Alarm` component as the macros `Alarm_init()` and `Alarm_dispatch()` to let the container initialize and dispatch events to the component, respectively. This approach insulates the container from the choice of the base class for the component. If later on I decide to derive `Alarm` from `QHsm`, for example, I need to change only the definitions of the `Alarm_init()` and `Alarm_dispatch()` macros, but I don't need to change the container code at all. Please note that the macros are unnecessary in the C++ implementation, because due to the compatibility between the `QHsm` and `QFsm` interfaces, the container state handler functions always dispatch events to the `Alarm` component in the same way by calling `me->alarm.dispatch()`.

Listing 5.6 Alarm component declaration (file `alarm.h`).

```
    #ifndef alarm_h
    #define alarm_h

    typedef struct AlarmTag {          /* the HSM version of the Alarm component */
        QFsm super;                                       /* derive from QFsm */
        uint32_t alarm_time;
    } Alarm;

    void Alarm_ctor(Alarm *me);
    #define Alarm_init(me_)            QFsm_init    ((QFsm *)(me_), (QEvent *)0)
    #define Alarm_dispatch(me_, e_)    QFsm_dispatch((QFsm *)(me_), e_)

    #endif                                                          /* alarm_h */
```

Listing 5.7 `Alarm` state machine definition (file `alarm.c`).

```
(1) #include "alarm.h"
(2) #include "clock.h"

    /* FSM state-handler functions */
(3) QState Alarm_initial(Alarm *me, QEvent const *e);
    QState Alarm_off    (Alarm *me, QEvent const *e);
    QState Alarm_on     (Alarm *me, QEvent const *e);

    /*..........................................................................*/
    void Alarm_ctor(Alarm *me) {
(4)     QFsm_ctor(&me->super, (QStateHandler)&Alarm_initial);
    }

    /* HSM definition ----------------------------------------------------------*/
    QState Alarm_initial(Alarm *me, QEvent const *e) {
        (void)e;               /* avoid compiler warning about unused parameter */
        me->alarm_time = 12*60;
        return Q_TRAN(&Alarm_off);
    }
    /*..........................................................................*/
    QState Alarm_off(Alarm *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                 /* while in the off state, the alarm is kept in decimal format */
(5)             me->alarm_time = (me->alarm_time/60)*100 + me->alarm_time%60;
                printf("*** Alarm OFF %02ld:%02ld\n",
                       me->alarm_time/100, me->alarm_time%100);
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                           /* upon exit, the alarm is converted to binary format */
(6)             me->alarm_time = (me->alarm_time/100)*60 + me->alarm_time%100;
                return Q_HANDLED();
            }
            case ALARM_ON_SIG: {
                return Q_TRAN(&Alarm_on);
            }
            case ALARM_SET_SIG: {
                           /* while setting, the alarm is kept in decimal format */
                uint32_t alarm = (10 * me->alarm_time
                                  + ((SetEvt const *)e)->digit) % 10000;
                if ((alarm / 100 < 24) && (alarm % 100 < 60)) {/*alarm in range?*/
                    me->alarm_time = alarm;
                }
                else {                          /* alarm out of range -- start over */
                    me->alarm_time = 0;
                }
                printf("*** Alarm SET %02ld:%02ld\n",
                       me->alarm_time/100, me->alarm_time%100);
                return Q_HANDLED();
            }
        }
        return Q_IGNORED();
    }
    /*..........................................................................*/
    QState Alarm_on(Alarm *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("*** Alarm ON %02ld:%02ld\n",
                       me->alarm_time/60, me->alarm_time%60);
                return Q_HANDLED();
            }
            case ALARM_SET_SIG: {
                printf("*** Cannot set Alarm when it is ON\n");
                return Q_HANDLED();
            }
            case ALARM_OFF_SIG: {
                return Q_TRAN(&Alarm_off);
            }
            case TIME_SIG: {
(7)             if (((TimeEvt *)e)->current_time == me->alarm_time) {
                    printf("ALARM!!!\n");
                           /* asynchronously post the event to the container AO */
(8)                 QActive_postFIFO(APP_alarmClock, Q_NEW(QEvent, ALARM_SIG));
                }
                return Q_HANDLED();
            }
        }
        return Q_IGNORED();
    }
```

(1,2) The `Alarm` component needs both the `alarm.h` interface and the `clock.h` container interface.
(3) The non-hierarchical state handler functions don't return the superstate (see Section 3.6 in Chapter 3).
(4) The `Alarm` constructor must invoke the constructor of its base class.
(5) Upon the entry to the "off" state, the alarm time is converted to the decimal format, in which 12:05 corresponds to decimal 1205.
(6) Upon the exit from the "off" state, the alarm time is converted back to the binary format, in which 12:05 corresponds to 12*60+5 = 725.

Note: The guaranteed initialization and cleanup provided by the entry and exit actions ensures that the time conversion will always happen, regardless of the way the state "off" is entered or exited. In particular, the alarm time will be always represented in decimal format while in the "off" state and in binary format outside of the "off" state.

---

(7) The `Alarm` component keeps receiving the TIME event from the `AlarmClock` container. `AlarmClock` conveniently provides the `current_time` event parameter, which the `Alarm` component can directly compare to its `me->alarm_time` extended-state variable.
(8) When the `Alarm` component detects the alarm time, it notifies the container by posting an event to it. Here, I am using a global pointer `APP_alarmClock` to the container active objects. An often used alternative is to store the pointer to the container inside each component.

Listing 5.8 `AlarmClock` state machine definition (file `clock.c`).

```
    #include "qp_port.h"
    #include "bsp.h"
(1) #include "alarm.h"
(2) #include "clock.h"

    /*..........................................................................*/
    typedef struct AlarmClockTag {               /* the AlarmClock active object */

(3)     QActive super;                                     /* derive from QActive */

        uint32_t current_time;                     /* the current time in seconds */
        QTimeEvt timeEvt;         /* time event generator (generates time ticks) */
```

```
(4)     Alarm alarm;                                /* Alarm orthogonal component */
    } AlarmClock;

    void AlarmClock_ctor(AlarmClock *me);                          /* default ctor */
                                                 /* hierarchical state machine ... */
    QState AlarmClock_initial    (AlarmClock *me, QEvent const *e);
    QState AlarmClock_timekeeping(AlarmClock *me, QEvent const *e);
    QState AlarmClock_mode12hr   (AlarmClock *me, QEvent const *e);
    QState AlarmClock_mode24hr   (AlarmClock *me, QEvent const *e);
    QState AlarmClock_final      (AlarmClock *me, QEvent const *e);

    /*..........................................................................*/
    void AlarmClock_ctor(AlarmClock *me) {                         /* default ctor */
        QActive_ctor(&me->super, (QStateHandler)&AlarmClock_initial);
(5)     Alarm_ctor(&me->alarm);                     /* orthogonal component ctor */
        QTimeEvt_ctor(&me->timeEvt, TICK_SIG);        /* private time event ctor */
    }
    /* HSM definition ----------------------------------------------------------*/
    QState AlarmClock_initial(AlarmClock *me, QEvent const *e) {
        (void)e;               /* avoid compiler warning about unused parameter */
        me->current_time = 0;
(6)     Alarm_init(&me->alarm);       /* the initial transition in the component */
        return Q_TRAN(&AlarmClock_timekeeping);
    }
    /*..........................................................................*/
    QState AlarmClock_final(AlarmClock *me, QEvent const *e) {
        (void)me;          /* avoid the compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("-> final\nBye!Bye!\n");
                BSP_exit();                            /* terminate the application */
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState AlarmClock_timekeeping(AlarmClock *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                                                  /* periodic timeout every second */
                QTimeEvt_fireEvery(&me->timeEvt,
                                  (QActive *)me, BSP_TICKS_PER_SEC);
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                QTimeEvt_disarm(&me->timeEvt);
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&AlarmClock_mode24hr);
            }
            case CLOCK_12H_SIG: {
                return Q_TRAN(&AlarmClock_mode12hr);
            }
            case CLOCK_24H_SIG: {
                return Q_TRAN(&AlarmClock_mode24hr);
            }
            case ALARM_SIG: {
                printf("Wake up!!!\n");
                return Q_HANDLED();
            }
            case ALARM_SET_SIG:
            case ALARM_ON_SIG:
            case ALARM_OFF_SIG: {
                          /* synchronously dispatch to the orthogonal component */
 (7)            Alarm_dispatch(&me->alarm, e);
                return Q_HANDLED();
            }
            case TERMINATE_SIG: {
                return Q_TRAN(&AlarmClock_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState AlarmClock_mode24hr(AlarmClock *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("*** 24-hour mode\n");
                return Q_HANDLED();
            }
            case TICK_SIG: {
 (8)            TimeEvt pe;    /* temporary synchronous event for the component */

                if (++me->current_time == 24*60) {  /* roll over in 24-hr mode? */
                    me->current_time = 0;
                }
                printf("%02ld:%02ld\n",
                       me->current_time/60, me->current_time%60);
 (9)            ((QEvent *)&pe)->sig = TICK_SIG;
(10)            pe.current_time = me->current_time;
                          /* synchronously dispatch to the orthogonal component */
(11)            Alarm_dispatch(&me->alarm, (QEvent *)&pe);
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&AlarmClock_timekeeping);
    }
    /*..........................................................................*/
    QState AlarmClock_mode12hr(AlarmClock *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("*** 12-hour mode\n");
                return Q_HANDLED();
            }
            case TICK_SIG: {
                TimeEvt pe;    /* temporary synchronous event for the component */
                uint32_t h;                  /* temporary variable to hold hour */

                if (++me->current_time == 12*60) {  /* roll over in 12-hr mode? */
                    me->current_time = 0;
                }
                h = me->current_time/60;
                printf("%02ld:%02ld %s\n", (h % 12) ? (h % 12) : 12,
                       me->current_time % 60, (h / 12) ? "PM" : "AM");
                ((QEvent *)&pe)->sig = TICK_SIG;
                pe.current_time = me->current_time;

                          /* synchronously dispatch to the orthogonal component */
                Alarm_dispatch(&me->alarm, (QEvent *)&pe);
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&AlarmClock_timekeeping);
    }
```

(1,2) The `AlarmClock` container includes its own interface `clock.h` as well as all interfaces to the component(s) it uses.

(3) The `AlarmClock` state machine derives from the QF class `QActive` that combines a HSM, an event queue, and a thread of execution. The `QActive` class actually derives from `QHsm`, which means that `AlarmClock` also indirectly derives from `QHsm`.

(4) The container physically aggregates all the components.

(5) The container must explicitly instantiate the components (in C).

(6) The container is responsible for initializing the components in its top-most initial transition.

(7) The container is responsible for dispatching the events of interest to the components. In this line, the container simply dispatches the current event `e`.

NOTE: The container's thread does not progress until the `dispatch()` function returns. In other words, component state machine executes its RTC step in the container's thread. This type of event processing is called *synchronous*.

---

(8) The temporary `TimeEvt` object to be synchronously dispatched to the component can be allocated on the stack.

(9-10) The container synthesizes the `TimeEvt` object on-the-fly and provides the current time.

(11) The temporary event is directly dispatched to the component.

Consequences

The Orthogonal Component state pattern has the following consequences.

- It partitions independent islands of behavior into separate state machine objects. This separation is deeper than with orthogonal regions because the objects have both distinct behavior and distinct data.
- Partitioning introduces a container–component (also known as parent–child or master–slave) relationship. The container implements the primary functionality and delegates other (secondary) features to the components. Both the container and the components are state machines.
- The components are often reusable with different containers or even within the same container (the container can instantiate more than one component of a given type).
- The container shares its execution thread with the components.
- The container communicates with the components by directly dispatching events to them. The components notify the container by posting events to it, never through direct event dispatching.
- The components typically use the Reminder state pattern to notify the container (i.e., the notification events are invented specifically for the internal communication and are not relevant

externally). If there are more components of a given type, then the notification events must identify the originating component (the component passes its ID number in a parameter of the notification event).

- The container and components can share data. Typically, the data is a data member of the container (to allow multiple instances of different containers). The container typically grants friendship to the selected components.
- The container is entirely responsible for its components. In particular, it must explicitly trigger initial transitions in all components[3], as well as explicitly dispatch events to the components. Errors may arise if the container "forgets" to dispatch events to some components in some of its states.
- The container has full control over the dispatching of events to the components. It can choose not to dispatch events that are irrelevant to the components. It can also change event types on the fly and provide some additional information to the components.
- The container can dynamically start and stop components (e.g., in certain states of the container state machine).
- The composition of state machines is not limited to just one level. Components can have state machine subcomponents; that is, the components can be containers for lower level subcomponents. Such a recursion of components can proceed arbitrarily deep.

Known Uses

The Orthogonal Component state pattern is popular in GUI systems. For example, dialog boxes are the containers that aggregate components in the form of dialog controls (buttons, check boxes, sliders, etc.). Both dialog boxes and dialog controls are event-driven objects with state behavior (e.g., a button has depressed and released states). GUIs also use the pattern recursively. For instance, a custom dialog box can be a container for the standard File-Select or Color-Select dialog boxes, which in turn contain buttons, check boxes, and so on.

The last example points to the main advantage of the Orthogonal Component state pattern over orthogonal regions. Unlike an orthogonal region, you can reuse a reactive component many times within one application and across many applications.

[3] In C, the container also must explicitly instantiate all components explicitly by calling their "constructors".