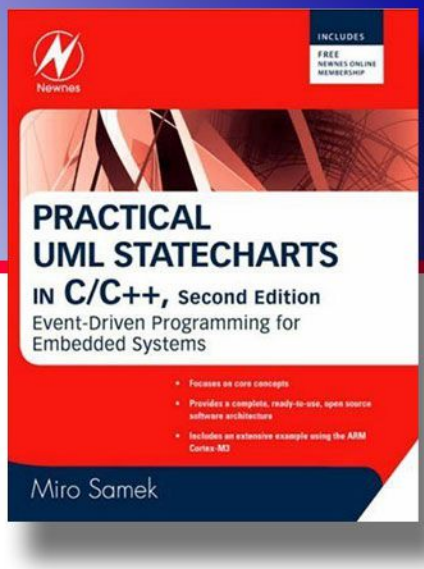
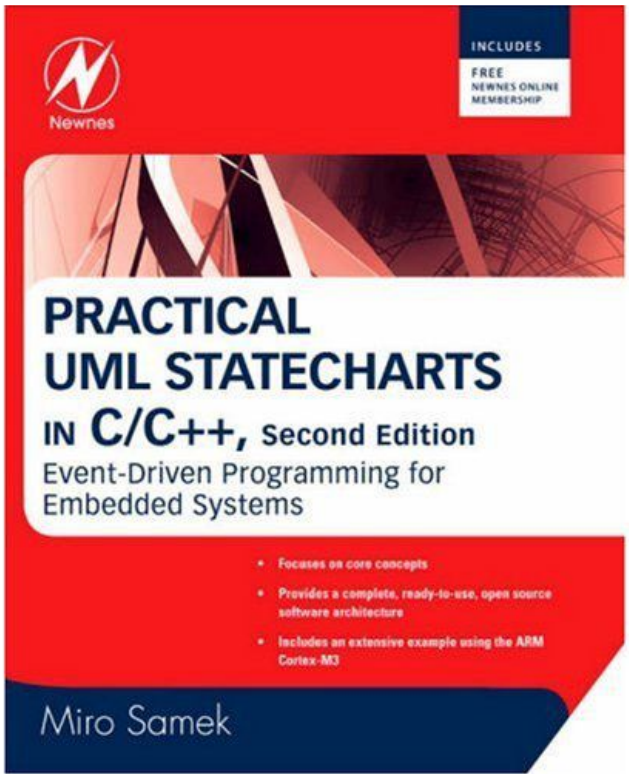
Quantum™ LeaPs
innovating embedded systems



Design Pattern
Transition to History

Document Revision C
October 2013

www.quantum-leaps.com
www.state-machine.com



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

Transition to History

Intent

Transition out of a composite state, but remember the most recent active substate so you can return to that substate later.

Problem

State transitions defined in high-level composite states often deal with events that require immediate attention; however, after handling them, the system should return to the most recent substate of the given composite sate.

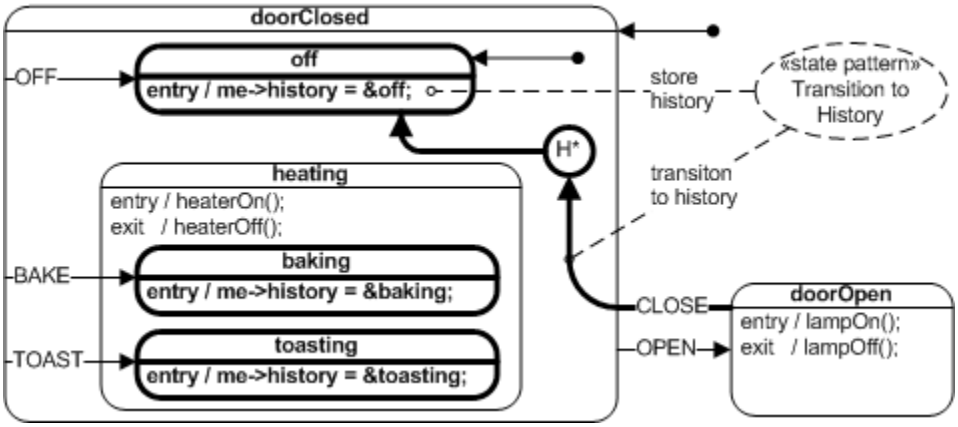
For example, consider a simple toaster oven. Normally the toaster operates with its door closed. However, at any time, the user can open the door to check the food or to clean the oven. Opening the door is an interruption; for safety reasons, it requires shutting the heater off and lighting an internal lamp. However, after closing the door, the toaster oven should resume whatever it was doing before the door was opened. Here is the problem: What was the toaster doing just before the door was opened? The state machine must remember the most recent state configuration that was active before opening the door in order to restore it after the door is closed again.

UML statecharts address this situation with two kinds of history pseudostates: shallow history and deep history (see Section 2.3.12 in Chapter 2). This toaster oven example requires the deep history mechanism (denoted as the circled H* icon in Figure 5.11). The QEP event processor does not support the history mechanism automatically for all states because it would incur extra memory and performance overheads. However, it is easy to add such support for selected states.

Solution

Figure 5.11 illustrates the solution, which is to store the most recently active *leaf* substate of the "doorClosed" state in the dedicated data member `doorClosed_history` (abbreviated to `history` in Figure 5.11). Subsequently, the transition to history of the "doorOpen" state (transition to the circled H*) uses the attribute as the target of the transition.

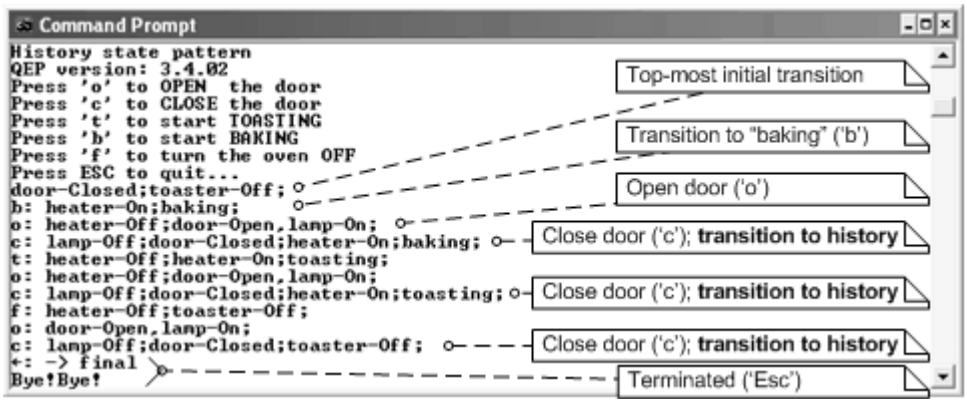
Figure 5.12 The Transition to History state pattern.



Sample Code

The sample code for the Transition to History state pattern is found in the directory `qpc\examples\win32\mingw\history\`. You can execute the application by double-clicking on the file `HISTORY.EXE` file in the `dbg\` subdirectory. Figure 5.13 shows the output generated by the `HISTORY.EXE` application. The application prints the actions as they occur. The legend of the key-strokes at the top of the screen describes how to generate events for the application. For example, you open the door by typing 'o', and close the door by typing 'c'.

Figure 5.13 Annotated output generated by `HISTORY.EXE`.



Listing 5.13 The Ultimate Hook sample code (file `hook.c`).

```
(1) #include "qep_port.h"

    /*..........................................................................*/
    enum ToasterOvenSignals {
        OPEN_SIG = Q_USER_SIG,
        CLOSE_SIG,
        TOAST_SIG,
        BAKE_SIG,
        OFF_SIG,
        TERMINATE_SIG                              /* terminate the application */
    };
    /*..........................................................................*/
    typedef struct ToasterOvenTag {
        QHsm super;                                          /* derive from QHsm */
(2)     QStateHandler doorClosed_history;    /* history of the doorClosed state */
    } ToasterOven;

    void ToasterOven_ctor(ToasterOven *me);                     /* default ctor */

    QState ToasterOven_initial   (ToasterOven *me, QEvent const *e);
    QState ToasterOven_doorOpen  (ToasterOven *me, QEvent const *e);
    QState ToasterOven_off       (ToasterOven *me, QEvent const *e);
    QState ToasterOven_heating   (ToasterOven *me, QEvent const *e);
    QState ToasterOven_toasting  (ToasterOven *me, QEvent const *e);
    QState ToasterOven_baking    (ToasterOven *me, QEvent const *e);
    QState ToasterOven_doorClosed(ToasterOven *me, QEvent const *e);
    QState ToasterOven_final     (ToasterOven *me, QEvent const *e);

    /*..........................................................................*/
    void ToasterOven_ctor(ToasterOven *me) {                    /* default ctor */
        QHsm_ctor(&me->super, (QStateHandler)&ToasterOven_initial);
    }

    /* HSM definitio ----------------------------------------------------------*/
    QState ToasterOven_initial(ToasterOven *me, QEvent const *e) {
        (void)e;               /* avoid compiler warning about unused parameter */
(3)     me->doorClosed_history = (QStateHandler)&ToasterOven_off;
        return Q_TRAN(&ToasterOven_doorClosed);
    }
    /*..........................................................................*/
    QState ToasterOven_final(ToasterOven *me, QEvent const *e) {
        (void)me;              /* avoid compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("-> final\nBye!Bye!\n");
                _exit(0);
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState ToasterOven_doorClosed(ToasterOven *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("door-Closed;");
                return Q_HANDLED();
```

```
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&ToasterOven_off);
            }
            case OPEN_SIG: {
                return Q_TRAN(&ToasterOven_doorOpen);
            }
            case TOAST_SIG: {
                return Q_TRAN(&ToasterOven_toasting);
            }
            case BAKE_SIG: {
                return Q_TRAN(&ToasterOven_baking);
            }
            case OFF_SIG: {
                return Q_TRAN(&ToasterOven_off);
            }
            case TERMINATE_SIG: {
                return Q_TRAN(&ToasterOven_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*.....................................................................*/
    QState ToasterOven_off(ToasterOven *me, QEvent const *e) {
        (void)me;              /* avoid compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("toaster-Off;");
(4)             me->doorClosed_history = (QStateHandler)&ToasterOven_off;
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&ToasterOven_doorClosed);
    }
    /*.....................................................................*/
    QState ToasterOven_heating(ToasterOven *me, QEvent const *e) {
        (void)me;              /* avoid compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("heater-On;");
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                printf("heater-Off;");
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&ToasterOven_doorClosed);
    }
    /*.....................................................................*/
    QState ToasterOven_toasting(ToasterOven *me, QEvent const *e) {
        (void)me;              /* avoid compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("toasting;");
(5)             me->doorClosed_history = (QStateHandler)&ToasterOven_toasting;
                return Q_HANDLED();
```

```
            }
        }
        return Q_SUPER(&ToasterOven_heating);
    }
    /*.....................................................................*/
    QState ToasterOven_baking(ToasterOven *me, QEvent const *e) {
        (void)me;              /* avoid compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("baking;");
(6)             me->doorClosed_history = (QStateHandler)&ToasterOven_baking;
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&ToasterOven_heating);
    }
    /*.....................................................................*/
    QState ToasterOven_doorOpen(ToasterOven *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("door-Open,lamp-On;");
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                printf("lamp-Off;");
                return (QState)0;
            }
            case CLOSE_SIG: {
(7)             return Q_TRAN(me->doorClosed_history); /* transition to HISTORY */
            }
        }
        return Q_SUPER(&QHsm_top);
    }
```

(1) Every QEP application needs to include `qep_port.h` (see Section 4.8 in Chapter 4).

(2) The `ToasterOven` state machine declares the history of the "doorClosed" state as a data member.

(3) The `doorClosed_history` variable is initialized in the top-most initial transition according to the diagram in Figure 5.12.

(4-6) The entry actions to all *leaf* substates of the "doorClosed" state record the history of entering those substates in the `doorClosed_history` variable. A leaf substate is a substate that has no further substates (see Section 2.3.8 in Chapter 2).

(7) The transition to history is implemented with the standard macro `Q_TRAN()`, where the target of the transition is the `doorClosed_history` variable.

Consequences

The Transition to History state pattern has the following consequences:

- It requires that a separate `QHsmState` pointer to state-handler function (history variable) is provided for each composite state to store the history of this state.
- The transition to history pseudostate (both deep and shallow history) is coded with the regular `Q_TRAN()` macro, where the target is specified as the history variable.

- Implementing the deep history pseudostate (see Section 2.3.12 in Chapter 2) requires explicitly setting the history variable in the entry action of each *leaf* substate of the corresponding composite state.
- Implementing the shallow history pseudostate (see Section 2.3.12 in Chapter 2) requires explicitly setting the history variable in each exit action from the desired level. For example, shallow history of the "doorClosed" state in Figure 5.12 requires setting `doorClosed_history` to `&ToasterOven_toasting` in the exit action from "toasting", to `&ToasterOven_baking` in the exit action from "baking", and so on for all direct substates of "doorClosed".
- You can explicitly clear the history of any composite state by resetting the corresponding history variable.

Known Uses

As a part of the UML specification, the history mechanism qualifies as a widely used pattern. The ROOM method [Selic+ 94] describes a few examples of transitions to history in real-time systems, whereas Horrocks [Horrocks 99] describes how to apply the history mechanism in the design of GUIs.

Summary

As Gamma and colleagues [GoF 95] observe: "*One thing expert designers know not to do is solve every problem from first principles.*" Collecting and documenting design patterns is one of the best ways of capturing and disseminating expertise in any domain, not just in software design.

State patterns are specific design patterns that are concerned with optimal (according to some criteria) ways of structuring states, events, and transitions to build effective state machines. In this chapter, I described just five such patterns and a few useful idioms for structuring state machines. The first two patterns, Ultimate Hook and Reminder, are at a significantly lower level than the rest, but they are so fundamental and useful that they belong in every state machine designer's bag of tricks.

The other three patterns (Deferred Event, Orthogonal Component, and Transition to History) are alternative, lightweight realizations of features supported natively in the UML state machine package [OMG 07]. Each one of these state patterns offers significant performance and memory savings compared to the full UML-compliant realization.