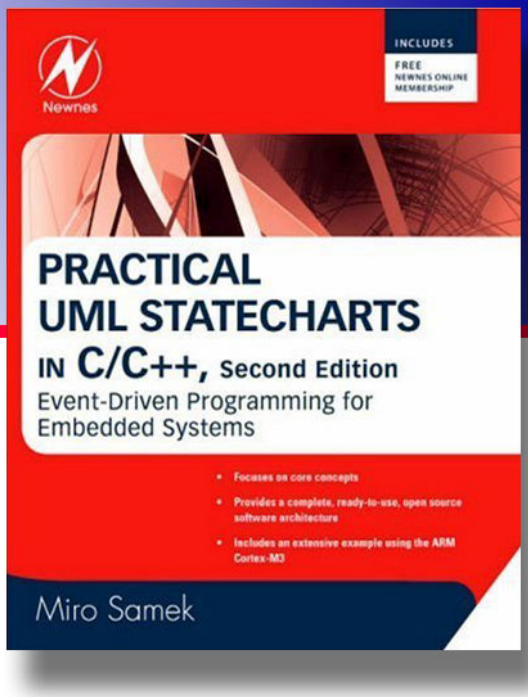
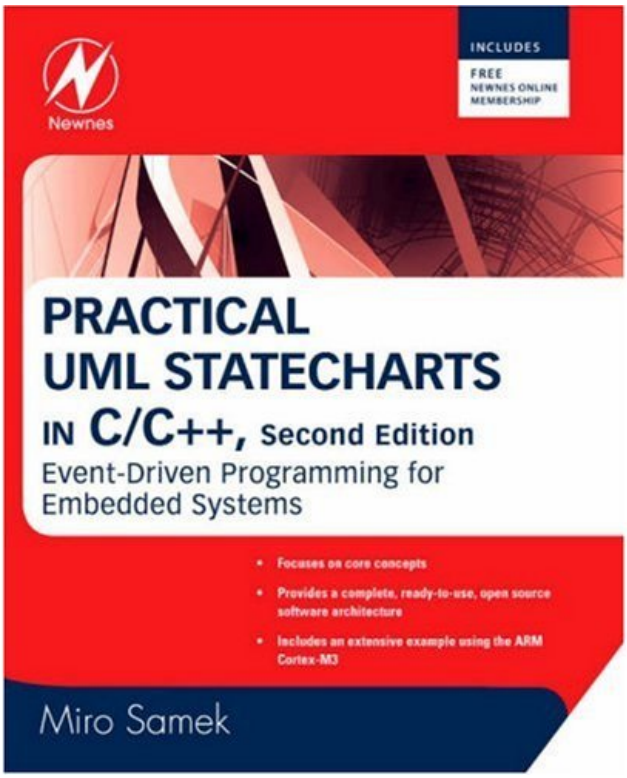
Quantum™ LeaPs
innovating embedded systems



Design Pattern
Ultimate Hook

Document Revision B
September 2008

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

Ultimate Hook

Intent

Provide common facilities and policies for handling events but let clients override and specialize every aspect of the system's behavior.

Problem

Many event-driven systems require consistent policies for handling events. In a GUI design, this consistency is part of the characteristic look and feel of the user interface. The challenge is to provide such a common look and feel in system-level software that client applications can use easily as the default. At the same time, the clients must be able to override every aspect of the default behavior easily if they so choose.

Solution

The solution is to apply programming-by-difference or, specifically in this case, the concept of hierarchical state nesting. A composite state can define the default behavior (the common look and feel) and supply an "outer shell" for nesting client substates. The semantics of state nesting provides the desired mechanism of handling all events, first in the context of the client code (the nested state) and of automatically forwarding of all unhandled events to the superstate (the default behavior). In that way, the client code intercepts every stimulus and can override every aspect of the behavior. To reuse the default behavior, the client simply ignores the event and lets the superstate handle it (the substate inherits behavior from the superstate).

Figure 5.1 The Ultimate Hook state pattern.

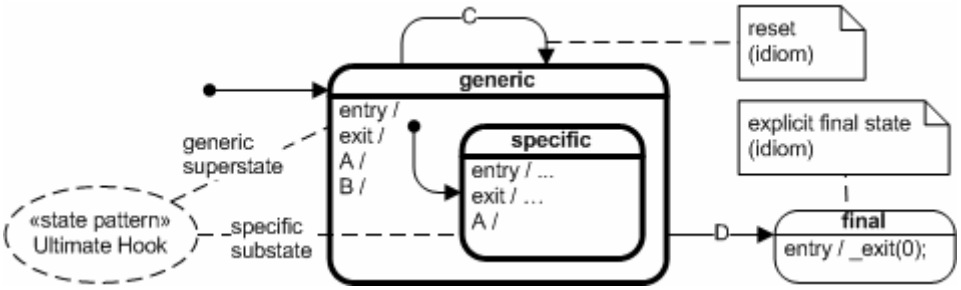


Figure 5.1 shows the Ultimate Hook state pattern using the collaboration notation adapted for states [OMG 07]. The dashed oval labeled «state pattern» indicates collaboration among states. Dashed arrows emanating from the oval indicate state roles within the pattern. States playing these roles are shown with heavy borders. For example, the state "generic" plays the role of the generic superstate of the pattern, whereas the state "specific" plays the role of the specific substate.

A diagram like this attempts to convey an abstract pattern but can only show a concrete example (instance) of the pattern. In this instance, the concrete "generic" state in Figure 5.1 handles events A and B as internal transitions, event C as a transition to self, and event D as the termination of the state machine. The concrete "specific" state overrides event A and provides its own initialization and cleanup (in entry and exit actions, respectively). Of course, another instance of the pattern can implement completely different events and actions.

A few idioms worth noting are illustrated in this state diagram. First is the overall canonical structure of the state machine that, at the highest level, consists of only one composite state (the pattern role of the generic superstate). Virtually every application can benefit from having such a highest level state because it is an ideal place for defining common policies subsequently inherited by the whole (arbitrary complex) submachine.

Note:	As described in Section 2.3.2 in Chapter 2, every UML state machine is a submachine of an implicit top state and so has the canonical structure proposed here. However, because you cannot override the top state, you need another highest level state that you can customize.

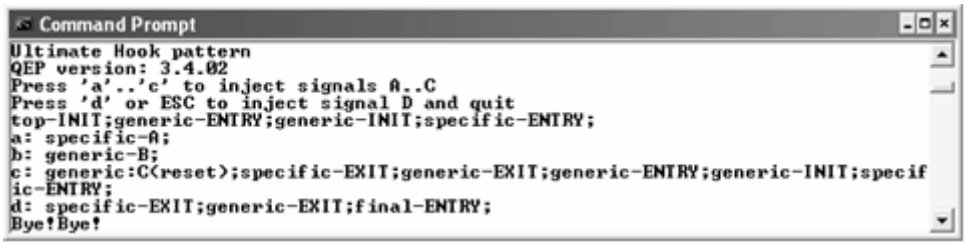
Within such a canonical structure, a useful idiom for resetting the state machine is an empty (actionless) transition to self in the "generic" superstate (transition C in Figure 5.1). Such a transition causes a recursive exit from all nested states (including the "generic" superstate), followed by initialization starting from the initial transition of the highest level state. This way of resetting a state machine is perhaps the safest because it guarantees proper cleanup through the execution of exit actions and clean initialization by entry actions and nested initial transitions. Similarly, the safest way to terminate a state machine is through an explicit transition out of the generic superstate to a "final" state (transition D in Figure 5.1) because all pertinent exit actions are executed. The QEP event processor does not provide a generic final state (denoted as the bull's eye in the UML). Instead, the statechart in Figure 5.1 proposes an idiom, which consists of an explicit state named "final" with an application-specific termination coded in its entry action[1].

Sample Code

The sample code for the Ultimate Hook state pattern is found in the directory `qpc\examples\80x86\dos\tcpp101\l\hook\`. You can execute the application by double-clicking on the file `HOOK.EXE` file in the `dbg\` subdirectory. Figure 5.2 shows the output generated by the `HOOK.EXE` application. Listing 5.1 shows the example implementation of the Ultimate Hook pattern from Figure 5.1.

[1] The calculator HSM designed in Chapter 2 and coded in Chapter 4 provides an example of the canonical state machine structure that uses the idioms to reset and terminate.

Figure 5.2 Output generated by HOOK.EXE.



Listing 5.1 The Ultimate Hook sample code (file hook.c).

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

    typedef struct UltimateHookTag {              /* UltimateHook state machine */
(2)     QHsm super;                                        /* derive from QHsm */
    } UltimateHook;

    void   UltimateHook_ctor    (UltimateHook *me);                    /* ctor */

(3) QState UltimateHook_initial (UltimateHook *me, QEvent const *e);
    QState UltimateHook_generic (UltimateHook *me, QEvent const *e);
    QState UltimateHook_specific(UltimateHook *me, QEvent const *e);
    QState UltimateHook_final   (UltimateHook *me, QEvent const *e);

(3) enum UltimateHookSignals {                         /* enumeration of signals */
        A_SIG = Q_USER_SIG,
        B_SIG,
        C_SIG,
        D_SIG
    };
    /*..........................................................................*/
    void UltimateHook_ctor(UltimateHook *me) {
        QHsm_ctor(&me->super, (QStateHandler)&UltimateHook_initial);
    }
    /*..........................................................................*/
    QState UltimateHook_initial(UltimateHook *me, QEvent const *e) {
        printf("top-INIT;");
        return Q_TRAN(&UltimateHook_generic);
    }
    /*..........................................................................*/
    QState UltimateHook_final(UltimateHook *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("final-ENTRY(terminate);\nBye!Bye!\n");
                exit(0);
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState UltimateHook_generic(UltimateHook *me, QEvent const *e) {
        switch (e->sig) {
```

```
            . . .
            case Q_INIT_SIG: {
                printf("generic-INIT;");
                return Q_TRAN(&UltimateHook_specific);
            }
            case A_SIG: {
                printf("generic-A;");
                return Q_HANDLED();
            }
            case B_SIG: {
                printf("generic-B;");
                return Q_HANDLED();
            }
            case C_SIG: {
                printf("generic-C(reset);");
(5)             return Q_TRAN(&UltimateHook_generic);
            }
            case D_SIG: {
(6)             return Q_TRAN(&UltimateHook_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*.....................................................................*/
    QState UltimateHook_specific(UltimateHook *me, QEvent const *e) {
        switch (e->sig) {
(7)         case Q_ENTRY_SIG: {
                printf("specific-ENTRY;");
                return Q_HANDLED();
            }
(8)         case Q_EXIT_SIG: {
                printf("specific-EXIT;");
                return Q_HANDLED();
            }
(9)         case A_SIG: {
                printf("specific-A;");
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&UltimateHook_generic);              /* the superstate */
    }
```

(1) Every QEP application needs to include `qep_porth.h` (see Section 4.8 in Chapter 4).
(2) The structure `UltimateHook` derives from `QHsm`.
(3) The `UltimateHook` declares the `initial()` pseudostate and three state handler functions: `generic()`, `specific()`, and `final()`.
(4) The signals A through D are enumerated.
(5) The transition-to-self in the "generic" state represents the reset idiom.
(6) The transition to the explicit "final" state represents the terminate idiom.
(7-8) The entry and exit actions in the "specific" state provide initialization and cleanup.
(9) The internal transition A in the "specific" state overrides the same transition in the "generic" superstate.

One option of deploying the Ultimate Hook pattern is to organize the code into a library that intentionally does not contain the implementation of the `UlimtateHook_specific()` state handler function. Clients would then have to provide their own implementation and link to the library to obtain the generic behavior. An

example of a design using this technique is Microsoft Windows, which requires the client code to define the `WinMain()` function for the Windows application to link.

Another option for the C++ version is to declare the `UlimtateHook::specific()` state handler as follows:

```
QState UltimateHook::specific(UltimateHook *me, QEvent const *e) {
    return me->v_specific(e);                              /* virtual call */
}
```

Where the member function `UltimateHook::v_specific(QEvent const *e)` is declared as a pure virtual member function in C++. This will force clients to provide implementation for the pure virtual state handler function `v_specific()` by subclassing the `UltimateHook` class. This approach combines behavioral inheritance with traditional class inheritance. More precisely, Ultimate Hook represents, in this case, a special instance of the Template Method design pattern [GoF 95].

Consequences

The Ultimate Hook state pattern is presented here in its most limited version — exactly as it is used in GUI systems (e.g., Microsoft Windows). In particular, neither the generic superstate nor the specific substate exhibits any interesting state machine topology. The only significant feature is hierarchical state nesting, which can be applied recursively within the specific substate. For example, at any level, a GUI window can have nested child windows, which handle events before the parent.

Even in this most limited version, however, the Ultimate Hook state pattern is a fundamental technique for reusing behavior. In fact, every state model using the canonical structure implicitly applies this pattern.

The Ultimate Hook state pattern has the following consequences:

- The specific substate needs to know only those events it overrides.
- New events can be added easily to the high-level generic superstate without affecting the specific substate.
- Removing or changing the semantics of events that clients already use is difficult.
- Propagating every event through many levels of nesting (if the specific substate has recursively nested substates) can be expensive.

The Ultimate Hook state pattern is closely related to the Template Method OO design pattern and can be generalized by applying inheritance of entire state machines.