Apparatus Framework

Cookbook for the usage of the main mechanisms

www.bbv.ch

Contents

- Task creation and usage
- Message passing mechanism
- Inter-task communication
- Observer pattern
- Monitor
- Mutexes, Semaphores
- Timers

Task

- Inherit from the base class and provide a runBody() method

```
#include "/ApparatusServices/KernelServices/exp/Task.hxx"
using namespace ApparatusFramework;

class PingPongTask : public Task
{
public:
        PingPongTask();
        void runBody();
};
```

The task is in the namespace ApparatusFramework. If you forget to declare this, the compiler won't find it.

A constructor to create the task (needed mandatory!).

This is the body of the task.

Task

- Task definitions

```
// Application Task definitions
const TaskId           PingPongTaskId_C           = 100;
const TaskPriority     PingPongTaskPriority_C     = 40;
const unsigned int     PingPongTaskStackSize_C    = 1000;
const unsigned int     PingPongTaskMsgQ_C         = 50;
const char             PingPongTaskName_C[]       = “tPingPong";
// Another Tasks definition
const TaskId           …
…
```

Unique. Starting point: free (could also start with 1)

The lower the number, the more important the task (0 highest)

In bytes (1k byte)

Maximum number of messages in the local task message queue.

Constant and unique

- The task definitions (ids etc.) should be defined previously by the architect and held in a single place (unique in the system).

Task

- Creation

```
PingPongTask::PingPongTask() :
    Task( PingPongTaskId_C,
          PingPongTaskPriority_C,
          0,
          PingPongTaskMsgQ_C,
          PingPongTaskName_C)
{
    startTask();
    …
}
```

Identifier

Priority

Stack size

Message queue size

Task name

Set task to ready

- Deletion or destruction of a task is never used in our system

bbv
behind things.

Task

- Task body

```
void PingPongTask::runBody()
{
    Msg* pMsg;
    PingPongMsg* pPmsg;

    for(;;) {
        pMsg = waitMsg();
        // global events as defined in the Harel theory
        switch(pMsg->m_msgId) {
            case MSG_PING_PONG:
                pPmsg = dynamic_cast<PingPongMsg*>(pMsg);
                assert(pPmsg != null)
                processPingPongMsg(pPmsg);
                break;
            default:
                break;
        }
        delete(pMsg);
    }
}
```

Implementation of task body

Never ending loop

Wait for an incoming message (event)

Cast to correct message (dynamic if possible)

Is possible if the dynamic cast fails

Handle the received message/event

IMPORTANT. DO NOT FORGET. Release the memory for the received message. Every entity that receives a message has to do this.

Task

- Define your own message type by inheriting from Msg
- You also need to define a unique id for this message.

```
#include “/ApparatusServices/KernelServices/exp/Msg.hxx”

class PingPongMsg : public Msg {
public:
   PingPongMsg(int pingPongData)  {m_msgId = MSG_PING_PONG;
                                   m_pingPongData = pingPongData;};
   Msg* clone(){return new PingPongMsg(*this);};
   int m_pingPongData;
};
```

Constructor initializes the message

Provide a clone method that does a deep copy of the message (needed for the observer pattern).

The message internal data that you want to transmit. Here it’s a public variable, you may want to make it private and provide setters and getters.

Task

- Sending a message to own task

```
void PingPongTask::sendPingPongMsg()
{
    int myData = 6;
    Msg *pMsg = new PingPongMsg(myData);
    sendMsg(this, pMsg);
}
```

Reserve (allocate) memory and initialize the message

Send message to own task

Task

- Sending a message to specific task

```
void AnotherTask::sendPingPongMsg()
{
    int myData = 6;
    Msg *pMsg = new PingPongMsg(myData)
    sendMsg(PostOffice::Default()->anotherTask(), pMsg);
}
```

Reserve (allocate) memory and initialize the message

Send message to this specific task

Fetches a pointer to another task in the system from a global object holding all task pointers.

Observer pattern (Observer)

- Create a new observer by inheriting from the base

The observer comes
from the pattern services

```
#include “/ApparatusServices/PatternServices/exp/Observer.hxx“
using namespace ApparatusFramework;

class PingPongObserver: public Observer<Msg*> {
public:
    PingPongObserver();
    virtual void update(Msg* msg,Observable<Msg*>* origin=0);
};
```

Initialize the template to
send Msgs

Provide an update method that is
called by the obervable whenever it
calls its dispatch method.

Observer pattern (Observable)

- Create a new observable by inheriting from one of the bases
  - Observable, generic template for an observable
  - KernelSimpleObservable, Observable that is able to handle kernel messages (Msg). Just one observer may attach.
  - KernelMultipleObservable, Same as simple, but multiple may attach

Observer pattern (Observable)

- Observer attachment to the observable

```
void PingPongObserver::initialize()
{
    // Register this object for message exchange (observer pattern)
    PostOffice::Default()->pingPongObservable()->attach(this);
    …
}
```

- The attachment is normally at startup time. Although the Observable provides a detach method, the mechanism is not built for dynamic attaches and detaches to an observable. This may lead to race condition errors in a multitasking environment. If you want to detach after startup, overwrite the attach() and detach() methods and protect them with a monitor.

Observer pattern (Observable)

- Sending notification to an observer (state of observable has changed)

Implementation of the notification dispatch routine

```
void PingPongObservable::dispatchPingPongMsg()
{
    Msg* pMsg = new PingPongMsg(6);
    dispatch(pMsg);
}
```

Reserve (allocate) memory for the notification (same mechanism as for sending of messages over tasks).

Dispatch notification to all registered observer

Observer pattern (Observer)

- Receiving notification from an observable

```
void PingPongObserver::update(Msg* pMsg){
    PingPongMsg* pPmsg;
    switch(pMsg->m_msgId) {
        case MSG_PING_PONG:
            pPmsg = dynamic_cast<PingPongMsg*>(pMsg);
            processPingPongMsg(pPmsg);
            break;
        …
        default:
            break;
    }
    delete(pMsg);
}
```

Implementation of the notification routine

Identify the message

Handle the notification message (event)

IMPORTANT! Make sure this message is handled fast or the task of the caller will be blocked!!! If you are a task, you may send the message to yourself and handle it in this context (if you send it further, do not delete it at the end of the update or make a new one!!!).

Release the memory for the notification message. Everyone who receives a message does that

Monitor

- Secures data and methods of a class in a multitasking environment

```
#include “/ApparatusServices/KernelServices/exp/Monitor.hxx”
using namespace ApparatusFramework;


class MonitorConsole : private Monitor {
public:
   void writeToConsole(const char* text, unsigned int aValue);
};

MonitorConsole writeToConsole(const char* text, unsigned int aValue)
{
   monitorEntry();
   printf("%s%d\n",text,aValue);
   monitorExit();}
```

Like the monitor were a member of this class (private inheritance)

Enter the protected area (another, following task that arrives here will block until the first one exits the monitor)

It is guaranteed that only one task will be here at one time.

Mutexes, Semaphores

- Synchronization using the KernelServices

```
#include “/ApparatusServices/KernelServices/exp/Mutex.hxx”
#include “/ApparatusServices/KernelServices/exp/Semaphore.hxx”
using namespace ApparatusFramework;
…
Mutex aMutex;
Semaphore aSemaphore(4);

aMutex.get();
aMutex.put();

aSemaphore.get();
aSemaphore.put();
…
```

After initialization, the mutex is released (you may get it).

Initializes the counting semaphore to 4.

- Be sure not to create the mutex or semaphores before the Kernel & OS have been started. Static or global objects are risky here.

Timers

- Creation

```
#include “/ApparatusServices/PatternServices/exp/TimerMgr.hxx”
using namespace ApparatusFramework;


…
    TimerMgr::TimerId timerId1;
    TimerMgr::TimerId timerId2;

    TimerMgr::Default()->createTimer(400, &timerId1);
    TimerMgr::Default()->createTimer(800, &timerId2);
…
```

The timer returns a unique timer id that identifies the timer.

Creates a timer with a duration of 800ms.

Timers

- Receiving

```
#include “/ApparatusServices/PatternServices/exp/TimerMgr.hxx”
using namespace ApparatusFramework;


void TimerTask::runBody()
{
    TimerMsg* pTMsg=0;
    for (;;) {
        pMsg = waitMsg();
        if(pMsg->m_msgId == MSG_TIMER_MGR_ID) {
           if(pTMsg = exampleCast(pTMsg,pMsg)) {
              if(pTMsg->m_theTimerId==timerId1) {
                   processTimer1();
              }
              if(pTMsg->m_theTimerId==timerId2) {
                   processTimer2();
              }
        } else {
            Logger::getLogger()->printLogText("That was not a timermessage !\n");
        }
        delete(pMsg);
    }
}
```

You h have to be a task in order to receive timer messages!

The id the timer manager reserves for its messages (each timer message will have this id).

Check the id to identify the timer that has fired.

Handle the timer.

Release the message.