Active object

Reviewer Approved

An active object is an object of a `CActive`-derived class. It uses an asynchronous service-provider to make an asynchronous service available to clients. An active object provides methods to issue a request to the service provider, get a notification when the request completes and to cancel the outstanding request. Below is an example of a simple active object.

MyActiveObject.h

```
#ifndef __MYACTIVEOBJECT_H__
#define __MYACTIVEOBJECT_H__
```

`CActive` class is declared in `e32base.h` so we need to include this header.

```
#include <e32base.h>
```

`CActive` class is Link against: `euser.lib` so we need to include this library to .mmp file.

```
LIBRARY euser.lib
```

Observer class to handle the async request result.

```
class MMyActiveObjectObserver
{
public:
```

Pure virtual function that shall be implemented by every user of `CMyActiveObject` object. It is called when async service provider completes the request.

```
 virtual void HandleRequestCompleted(TInt aError) = 0;
};
```

Active object that utilizes the use of `RMyAsyncServiceProvider`.

```
class CMyActiveObject : public CActive
{
public:
 static CMyActiveObject* NewL(TInt aPriority);
 ~CMyActiveObject();

 void DoAsyncAction(MMyActiveObjectObserver* aObserver);

protected:
 // inherited from CActive
 void RunL();
 void DoCancel();
 TInt RunError(TInt aError);

private:
 CMyActiveObject(TInt aPriority);
 void ConstructL();

private:
 // Provide your own service provider object e.g. RTimer here
        RMyAsyncServiceProvider iServiceProvider;

 MMyActiveObjectObserver* iObserver; // Observer
};

#endif // __MYACTIVEOBJECT_H__
```

MyActiveObject.cpp

```
#include "myactiveobject.h"
```

A factory function to create our active object. See Two-phase construction. The active object's priority is passed as the param, see `CActive::TPriority` for possible priority values.

```
CMyActiveObject* CMyActiveObject::NewL(TInt aPriority)
        {
 CMyActiveObject* self = new (ELeave) CMyActiveObject(aPriority);
 CleanupStack::PushL(self);
 self->ConstructL();
 CleanupStack::Pop();
 return self;
        }
```

Constructor. It is called from our `NewL()` function. Here we set active object's priority.The priority can be changed later by using the method

(C) Copyright Nokia 2013. All rights reserved.

```
CMyActiveObject::CMyActiveObject(TInt aPriority)
: CActive(aPriority)
{
}
```

Initialize our active object.

```
void CMyActiveObject::ConstructL()
{
```

Add the active object to the Active Scheduler. If we do not do this we will end up with the E32User-CBase panic when our async request completes. `CActiveScheduler::Add()` method is leave-safe so actually it can be called in active object's constructor.

```
 CActiveScheduler::Add(this);
```

Any necessary code to initialize the async service provider, leave in case of an error. for example, if initialization is through a Connect() method that returns a TInt error:

```
 User::LeaveIfError(iServiceProvider.Connect());
}
```

Destructor.

```
CMyActiveObject::~CMyActiveObject()
{
```

`Cancel()` should be always called in active object's destructor to cancel an outstanding request if there is one. If there is no request pending then `Cancel()` just does nothing, but if we do not call `Cancel()` when having an outstanding request E32User-CBase panic 40 will be raised.

```
 Cancel();
```

Close the session to our async service provider (as you would do in any destructor).

```
 iServiceProvider.Close();
}
```

This method is called by our active object users to start an async action. The argument is a pointer to an observer object who's `HandleRequestCompleted()` method will be called when the async request completes.

```
void CMyActiveObject::DoAsyncAction(MMyActiveObjectObserver* aObserver)
{
```

Assert that we do not have an outstanding request already and panic if we have. If we do not do this checking then `SetActive()` will raise E32User-CBase 42 panic. See Panic for information about asserts and panics.

```
 __ASSERT_ALWAYS(!IsActive(), User::Panic(KMyActivePanic, EAlreadyActive));
```

Assert that aObserver is not NULL.

```
 __ASSERT_ALWAYS(aObserver, User::Panic(KMyActivePanic, ENoObserver));
```

Issue a request to our service provider. We pass our `iStatus` as the argument. The service provider sets our `iStatus` to `KRequestPending`. When the service provider finishes it's work Active Scheduler will complete the `iStatus` with the error code indicating if the operation was successful or not.

```
 iServiceProvider.DoService(iStatus);
```

Mark our active object as active which means that we are waiting for our outstanding request to complete.

```
 SetActive();
}
```

`RunL()` is called by the Active Scheduler when our request is completed. All active objects need to implement this function. Active Scheduler runs `RunL()` under a trap harness. If it leaves active object's `RunError()` is called.

```
void CMyActiveObject::RunL()
{
```

`iStatus` contains the error code indicating if our request completed successfully or not. We leave if there is an error.

```
 User::LeaveIfError(iStatus.Int());
```

If we reached here it means that our request completed successfully - let's tell the observer about it.

(C) Copyright Nokia 2013. All rights reserved.

```
 iObserver->HandleRequestCompleted(KErrNone);
}
```

Every active object shall implement this function. `DoCancel()` is called as part of the active object's `Cancel()` and shall cancel the outstanding request.

```
void CMyActiveObject::DoCancel()
{
 iServiceProvider.Cancel();
}
```

It is not necessary to implement `RunError()` function but it is very useful to do it. It is called by the Active Scheduler if a leave occurs in active object's `RunL()`. The overrided implementation should handle the error (if possible) and always return `KErrNone`. If `RunError()` is not implemented by the active object then the default version is called that just returns the leave code (`aError`). If `RunError()` returns anything but `KErrNone` Active Scheduler calls it's `Error()` function. If `CActiveScheduler::Error()` is not overriden by a `CActiveScheduler`-derived class which is usually not the case then the default implementation of `CActiveScheduler::Error()` raises E32USER-CBase 47 panic.

```
TInt CMyActiveObject::RunError(TInt aError)
{
```

Inform our observer about the error so that it can handle it like performing recovery actions or issuing another async request. `RunError()` should be leave-safe so we run `HandleRequestCompleted()` under a trap harness. A callback should never leave.

```
 iObserver->HandleRequestCompleted(aError);
 return KErrNone;
}
```

Self activation

Sometimes there is need for an active object to activate itself without waiting for any resource. This will result RunL to be called by active sceduler, when active scheduler has time for it. This is similar to using timer with timeout nearly zero.

```
  if( IsActive()) // cannot activate allready active object
    return;
  TRequestStatus * status = &iStatus;
  User::RequestComplete(status, 0);
  SetActive();
```

How is this different from calling RunL directly?

This will result callback to RunL, when active sceduler has time for it and not synchronously, like directly calling RunL.

Internal Links

Active Objects in Symbian OS

Responsibilities of Asynchronous Service Provider in Active Objects

External Links

CActive in Symbian OS Developer Library

CActiveScheduler in Symbian OS Developer Library

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.