

Enabling a MOOS Application for AppCasting

June 2018

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1	Overview	1
2	Sub-classing the AppCastingMOOSApp Superclass	2
3	Invoking Superclass Methods in the Iterate() Method	2
4	Invoking a Superclass Method in the OnNewMail() Method	3
5	Invoking a Superclass Method in the OnStartup() Method	3
6	Invoking a Superclass Method When Registering for Variables	3
7	Implementing a buildReport Method for Generating AppCasts	4
8	Posting Events	5
9	Posting Run Warnings	6
10	Posting Configuration Warnings	7

1 Overview

In this section we discuss the steps for enabling a new or existing MOOS application to support appcasting. Much of the requisite appcasting source code is the same for any application. This is captured in a new `AppCastingMOOSApp` class to minimize appcasting boilerplate code. This class, as indicated in Figure 1, is a subclass of the common `CMOOSApp` class distributed with core MOOS.

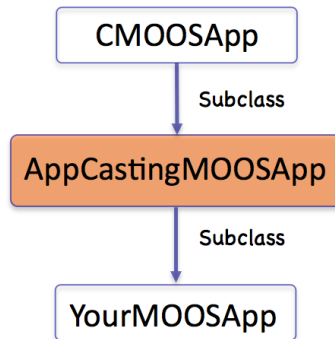


Figure 1: On-demand appcasting is implemented in a new `CMOOSApp` subclass called `AppCastingMOOSApp`.

The following is the complete list of required steps:

- Subclass the `AppCastingMOOSApp` superclass,
- Invoke a pair of superclass methods in the `Iterate()` function,
- Invoke a superclass method in the `OnNewMail()` function,
- Invoke a superclass method in the `OnStartup()` function,
- Invoke a superclass method when registering for variables,
- Implement a `buildReport()` function where appcasts are formed.

In our discussions to follow, a hypothetical `YourMOOSApp` application and class definition is described. The above steps are the minimum requirements for appcasting, and they are fairly boilerplate, in most cases a single line of code. To make good use of the appcasting features, configuration warnings, run warnings and events may be placed in your application code. This is neither mandatory nor boilerplate, but really dependent on the application. Nevertheless, a few rules of thumb discussed:

- Posting events,
- Posting run warnings,
- Posting configuration warnings.

[2]

2 Sub-classing the AppCastingMOOSApp Superclass

The first step is to make `YourMOOSApp` a subclass of the `AppCastingMOOSApp`. This brings your application class everything from the traditional `CMOOSApp` class as well as the appcasting features of the `AppCastingMOOSApp` class. The only additional thing besides declaring the superclass is to declare the `buildReport()` function. This virtual function is invoked when an appcast has been deemed warranted. Appcasts are not typically generated on each iteration. See the later discussion about *on-demand appcasting*. The contents of the `buildReport()` function are discussed in greater detail in Section 7.

Listing 2.1: Pseudocode for sub-classing the AppCastingMOOSApp superclass.

```
1 #include "MOOS/libMOOS/Thirdparty/AppCasting/AppCastingMOOSApp.h"
2
3 class YourMOOSApp : public AppCastingMOOSApp      // Instead of CMOOSApp
4 {
5     // All your normal class declaration stuff
6
7     bool buildReport();                            // Add this line
8 };
```

3 Invoking Superclass Methods in the Iterate() Method

The next step is to implement `YourMOOSApp::Iterate()` to invoke two superclass functions; one at the very beginning and one at the very end. The first superclass function, on line 2 below, does certain

common bookkeeping such as incrementing the counter representing the number of application iterations, and updating a variable holding the present MOOS time. The second superclass function, on line 6, invokes the on-demand appcasting logic discussed previously. If an appcast is deemed warranted, it will invoke the `buildReport()` function.

Listing 3.2: Pseudocode for invoking subclass methods in the `Iterate()` method.

```
1 bool YourMOOSApp::Iterate()
2 {
3     AppCastingMOOSApp::Iterate();           // Add this line
4
5     // Do all your normal Iterate stuff
6
7     AppCastingMOOSApp::PostReport();        // Add this line
8     return(true);
9 }
```

4 Invoking a Superclass Method in the `OnNewMail()` Method

The next step is to implement `YourMOOSApp::OnNewMail()` to invoke a superclass function to have the first opportunity to handle incoming mail. For example, `APPCAST_REQ` mail is handled in the superclass. The list of mail messages is passed by reference to the superclass handler, allowing the `AppCastingMOOSApp::OnNewMail()` function to remove handled messages before returning to the mail handling implemented in `YourMOOSApp::OnNewMail()`.

Listing 4.3: Pseudocode for invoking a superclass method in the `OnNewMail()` method.

```
1 bool YourMOOSApp::OnNewMail(MOOSMSG_LIST &NewMail)
2 {
3     AppCastingMOOSApp::OnNewMail(NewMail); // Add this line
4
5     // Do all your other normal mail handling.
6 }
```

5 Invoking a Superclass Method in the `OnStartup()` Method

The next step is to implement `YourMOOSApp::OnStartup()` to invoke a superclass function to perform startup steps needed by the `AppCastingMOOSApp` superclass.

Listing 5.4: Pseudocode for invoking a superclass method in the `OnStartup()` method.

```
1 void YourMOOSApp::OnStartup()
2 {
3     AppCastingMOOSApp::OnStartup();        // Add this line
4
5     // Do all your other startup stuff
6 }
```

6 Invoking a Superclass Method When Registering for Variables

The next step is to invoke the `AppCastingMOOSApp::RegisterVariables()` wherever variables are registered in `YourMOOSApp` implementation. Many application developers have, in practice, created a

dedicated `registerVariables()` function, typically invoked at the conclusion of both `OnStartup()` and `OnConnectToServer()`. The following example is one way to handle this.

Listing 6.5: Pseudocode for invoking a superclass method when registering for variables.

```
1 void YourMOOSApp::registerVariables()
2 {
3     AppCastingMOOSApp::RegisterVariables();    // Add this line
4
5     // Do all your other registrations
6 }
```

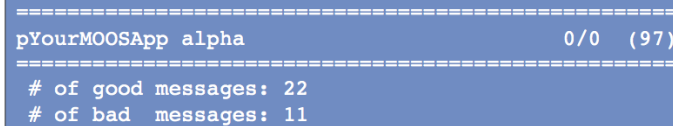
7 Implementing a `buildReport` Method for Generating AppCasts

The `buildReport()` function is where appcasts are made! The action that happens here is unique to the application. The form is designed by the application developer to reflect the most meaningful, concise snapshot of the application's present status. Recall that it is invoked automatically when or if the application deems an appcast is to be generated. For the purposes here though, a decision has indeed been made to generate an appcast, and `buildReport()` has been invoked to see that it happens. A simple example of `buildReport()` is shown below in Listing 6

Listing 7.6: Pseudocode for a very simple `buildReport()` example.

```
1 bool YourMOOSApp::buildReport()
2 {
3     m_msgs << "Number of good messages: " << m_good_message_count << endl;
4     m_msgs << "Number of bad messages: " << m_bad_message_count << endl;
5
6     return(true);
7 }
```

This simple example would generate something similar to the appcast rendered in Figure 2.



```
=====
pYourMOOSApp alpha                                0/0  (97)
=====
# of good messages: 22
# of bad messages: 11
```

Figure 2: The rendering of a very simple appcast with just two message lines, no warnings, and no events. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Assume for the sake of the example that the two counter variables at the end of lines 2 and 3 are member variables for the fictitious `YourMOOSApp` class. The member variable `m_msgs` however is an STL stringstream also declared in the `AppCastingMOOSApp` class, for holding message output. The primary means of building an appcast is to add successive lines to the appcast via:

```
m_msgs <<    <element>    << endl;
```

where <element> may be a string, double, int, unsigned int, or any combination joined by the "<<" operator. A new line is indicated by tacking on the newline, "\n", at the end. That's pretty much it, but there are a few other noteworthy points:

- Run warnings, configuration warnings, and events are not added during `buildReport()`, though not strictly prevented. They are more typically added as warnings are discovered, or events occur during the normal mail handling or iterate cycle.
- The header lines shown in Figure 2 is made automatically by the uMAC tool. They grab information in the appcast such as the application name and iteration number. These are filled by the boilerplate function calls such as `AppCastingMOOSApp::Iterate()` described earlier.
- The appcast is automatically cleared prior to each invocation of `buildReport()`. Warnings and events however are not cleared as discussed earlier.
- Returning true indicates that the appcast was indeed populated. Returning false would result in the appcast not being sent to the terminal or published to the MOOSDB. This is useful for some applications that may want to apply additional criteria before deciding to appcast.
- Although report formatting, e.g., columns or tables, is not natively supported somehow in the `buildReport()` routine, there are tools available that facilitate formatting that work easily with the `buildReport()` interface. They are discussed later in Section 99.

8 Posting Events

Events are messages (strings) that the application developer deems to be noteworthy enough to want to include in appcast output. An event may be posted anywhere in the application code with the `reportEvent()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportEvent("Good msg received: " + message);
```

When the appcast is rendered in either a uMAC tool or in the terminal, the result would look similar to that in Figure 3.

```
=====
pYourMOOSApp alpha                                0/1 (97)
=====
RunTime Warnings: 14
 [11] Bad msg received: bricks

# of good messages: 1
# of bad  messages: 14

=====
Most Recent Events (22):
=====
[23.09] Good msg received: happyjoy
```

Figure 3: The rendering of a simple appcast with two message lines, a single run warning, and single event. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Recall that only a limited number of events are retained. Older events are dropped once a maximum amount is exceeded. The default event list size is eight, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_events = 25
```

The event list size may be set to at most 100.

9 Posting Run Warnings

Run warnings are similar to events, but they convey that something may have gone wrong. A run warning may be posted anywhere in the application code with the `reportRunWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportRunWarning("Bad msg received: " + message);
```

The appcast structure and uMAC tools are implemented such that run warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a run warning, the uMAC utilities will indicate so by turning the text red, as in Figure 4. Even when the focus of the uMAC utility is not on the particular appcast containing the run warning, the menu items for the appcast and vehicle will be rendered red. In Figure 4 for example, the menu browser focus is on vehicle `archie` and the `uFldMessageHandler` application. The red highlights also indicate there is a run warning on another application on `archie`, and there is also a run warning on vehicle `charlie`.
- An appcast request to an application may specify the reporting threshold to be "run_warning". In this case an application will repeatedly choose not to publish an appcast unless a new run warning has been generated.
- The uMAC tools also keep a running tally of run warnings for each vehicle and each application under the column labeled "RW" as shown in Figure 4.

File AppCasting							
Node	AC	CW	RW	App	AC	CW	RW
archie	200	0	1	pNodeReporter	97	0	0
shoreside	25	0	0	pBasicContactMgr	7	0	0
betty	26	0	0	uFldMessageHandler	82	0	0
david	28	0	0	pHelmIvP	3	0	0
prey	21	0	0	uProcessWatch	3	0	1
charlie	23	0	1	pHostInfo	4	0	0
ernie	28	0	0	uFldNodeBroker	4	0	0

Figure 4: The uMAC tools will highlight an application that has produced an appcast with a run warning. If the run warning occurred on a node not currently in focus, e.g., charlie in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the run warning.

Recall that only a limited number of run warnings are retained. Unlike events where the older ones are dropped once a maximum has been exceeded, old run warnings are never dropped. After the maximum has been reached, the generic warning "Other Run Warnings" is simply incremented. The default list size is ten, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_run_warnings = 50
```

The run warning list size may be set to at most 100.

10 Posting Configuration Warnings

Configuration warnings are similar to run warnings but they are typically only posted during the application startup, when the mission configuration file is read. A configuration warning may be posted with the `reportConfigWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportConfigWarning("Problem configuring FOOBAR. Expected a number but got: " + str);
```

There is a second way to post a configuration warning. This second method takes as an argument the original full configuration parameter line found in the mission file. Before posting the configuration warning it checks to see if the parameter was something that likely was handled by the superclass. This prevents the application from reporting that `AppTick=10` is an unknown parameter for example.

```
reportUnhandledConfigWarning(original_full_config_line);
```

The appcast structure and uMAC tools are implemented such that configuration warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a configuration warning, the uMAC utilities will indicate so by turning the text green, as in Figure 5. Even when the focus of the uMAC utility is not on the particular appcast containing the config warning, the menu items for the appcast and vehicle will be rendered green. In Figure 5 for example, the menu browser focus is on the `shoreside` and the `uTimerScript` application application. The green highlights indicate there is a configuration warning in the `uFldShoreBroker` application and on other applications on `archie`, `charlie`, and `ernie`.
- The uMAC tools also keep a running tally of configuration warnings for each vehicle and each application under the column labeled "CW" as shown in Figure 5.

File AppCasting									
Node	AC	CW	RW		App	AC	CW	RW	
-----	---	--	--		-----	---	--	--	
shoreside	102	1	0		uTimerScript	91	0	0	
david	16	0	0		uMACView	3	0	0	
prey	12	0	0		uFldNodeComms	2	0	0	
betty	16	0	0		pMarineViewer	2	0	0	
archie	16	2	0		pHostInfo	2	0	0	
charlie	16	1	0		uFldShoreBroker	2	1	0	
ernie	16	1	0						

Figure 5: The uMAC tools include will highlight an application that has produced an appcast with a configuration warning. If the configuration warning occurred on a node not currently in focus, e.g., archie, charlie, or ernie in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the warning.

Like run warnings and events, configuration warnings are limited in number to prevent runaway growth in the size of an appcast over time. The limit however is large, 100, and fixed. Presumably the number of configuration warnings is limited by the number of possible configuration parameters for an application, and large number of configuration warnings usually indicates that a mission should be halted and fixed before moving on.

The example code in Listing 7 below is an example `OnStartUp()` method showing the intended scenarios of reporting configuration warnings.

Listing 10.7: Pseudocode example for `OnStartUp()` configuration warning handling.

```

1  bool YourMOOSApp::OnStartUp()
2  {
3      AppCastingMOOSApp::OnStartUp();
4
5      STRING_LIST sParams;
6      if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
7          reportConfigWarning("No config block found for " + GetAppName());
8
9      STRING_LIST::iterator p;
10     for(p=sParams.begin(); p!=sParams.end(); p++) {
11         string orig = *p;
12         string line = *p;
13         string param = toupper(MOOSChomp(line, "="));
14         string value = line;
15
16         if(param == "FOO") {
17             bool handled = handleConfigFOO(value);
18             if(!handled)
19                 reportConfigWarning("Problem with configuring FOO: " + value);
20         }
21         else if(param == "BAR")
22             bool handled = handleConfigBAR(value);
23             if(!handled)
24                 reportConfigWarning("Problem with configuring BAR: " + value);
25     }
26
27     else
28         reportUnhandledConfigWarning(orig);

```



```
29     }  
30     return(true);  
31 }
```

There are a few issues worth noting in this example:

- A check is made that the application actually has a configuration block. This is done on lines 5-6, and catches a common bug with newly minted mission files.
- Checks are made that known parameters have legal values. This is done for the parameters `FOO` in lines 15-19 and `BAR` in lines 20-24 in Listing 6. In each case an external handler is invoked, e.g., `handleConfigFOO(value)` on line 16, which returns a Boolean indicating whether the parameter value was proper or not. If not, a configuration warning is reported as on lines 18 and 23.
- A final case is handled (lines 26-27) if the present parameter is not matched by any of the previous cases. This catches the common mistake of mis-spelling the parameter name. The `reportUnhandledConfigWarning()` function is used rather than the `reportConfigWarning()` function. The former function takes the whole original configuration line as input and checks to see if the parameter was a parameter handled at the superclass level. This prevents the generation of a warning for a line like `AppTick=5`.