



Mission Director – Beginners' Guide

Introduction.....	3
Mission Director - setting up	4
XML editing tools.....	5
Creating a new Mission Director mission file	6
XML Formatting requirements.....	7
< cues >	7
< cue >	8
< condition >	8
Variables.....	9
< timing >.....	11
Units of Time and Distance.....	12
Expressions	12
< action >.....	13
Example 1.....	14
Testing your first mission file.....	15
'In Space' mission-writing tutorial	15
BBS mission-writing tutorial	42
A variable worth knowing about	48
Incorporating In-Game sounds	50
Incorporating texts	50
Testing your missions	51
Glossary	53
Appendices	54
Appendix 1: Instantiation.....	54
Appendix 2: Library Cues	55

INTRODUCTION

In previous games of the X Series, missions (both plot and non-plot) have been tricky to develop, difficult to test and prone to a high error rate, due to the strict coding requirements of KC. This resulted in show-stopping plot bugs and a very limited variety of non-plot missions.

The aim of the Mission Director is to provide a user-friendly method of developing and scripting missions, which is easier and more approachable for the non-programmer. A secondary aim is to make it possible to deliver mission-based content without the need for game patches.

An **XML**-based mission-design framework has been selected to provide easy 'plug-in' functionality and more accessibility to non-programming mission designers. That's where you come in...

With a basic understanding you can easily produce events in the game that can collectively form the features of a mission for the player to complete, while interacting with any game object, like stations or ships.

It is relatively easy to get started with Mission Director and it requires only a very rudimentary understanding of 'data logic' in order to build missions. As your confidence in using the Mission Director grows, so too will the complexity of the 'code' and therefore the missions that you will be able to develop.

The structure of mission files is fairly rigid and lends itself well to simpler debugging, testing and modification without fear of damaging underlying game code. You can easily produce a simple event, get to see and try it out in the game, change it again and then see these changes just by resetting the Mission Director in-game or reloading the game itself. This unprecedented flexibility is one of the key features of the Mission Director and speeds up the mission development process remarkably.

Examples

A range of missions is already being produced as MD development progresses. A great way to understand how things work is to look at an example of one of these and keep editing it with changes to see what the effects are. This will not only help you with understanding how things are done, but will also provide you with a library of examples of good practice and guidance to achieve what you want to.

It will help even more, if you study a mission similar to the one you are developing, so you can change, edit or even take slices of code for your own mission. Doing so is not plagiarism, it is a compliment to the original developer. This guide will give you a basic introduction to the key elements of a mission and will then progress to a couple of mission tutorials, providing a step by step guide to creating a mission. Once you have completed the tutorials you should have a better idea of how the system works and get your mission development skills off to a good start.

Scope

While some experience of programming can be useful for understanding processes, it's by no means essential. In fact it can sometimes be a hindrance, as MD is event-driven, rather than object-driven, so programmers will have to think differently. This guide assumes you have very limited or no knowledge of programming and has been written for beginners. It is assumed, however, that you have a good knowledge and experience of the X-Universe. It is hoped that it is useful to all who choose to use it.

Disclaimer

At this stage of its development this version of the Mission Developer is unsupported by Egosoft. Assistance can be obtained from volunteers on the forum, but any problems with coding or schemas will not be fixed. There are a number of known minor issues, but none of these should significantly affect your use of the Mission Director.

MISSION DIRECTOR - SETTING UP

The Mission Director is activated concurrently with the Script Editor when the player name is changed to Thereshallbewings. You need to ensure that you have a folder in your main game folder called 'director'. You need only have the mission file you want to play in this folder, but if you wish to edit files, you should have the files below.

Location of MD files

director - the files required to edit mission files are illustrated in the upper portion of Figure 1:

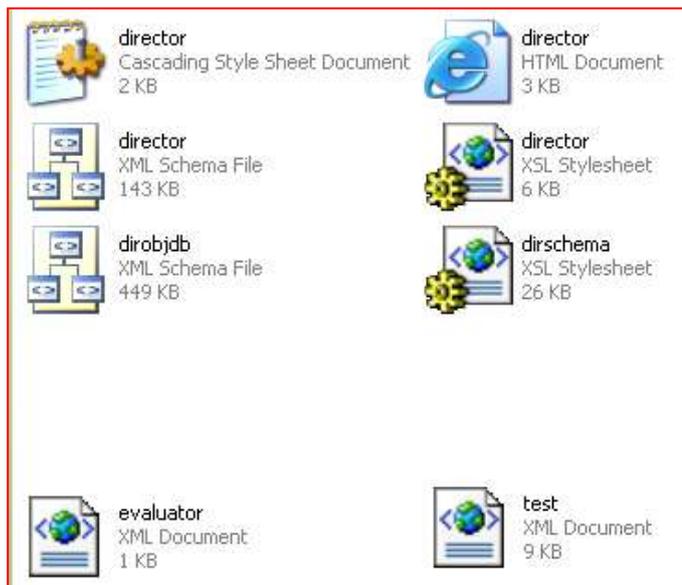


Figure 1 'Deepsilver\X3 Reunion\director' folder

The six files at the top of Figure 1 **must** all be present in the *director* folder if you wish to write/edit missions. Any number of XML mission files can exist in the directory and the game will attempt to load all XML files into the game when it loads. In order to ensure there are no conflicts, it is important that all cues in these files are uniquely identified. The two other files are example XML files.

Here's an overview of the function of some of those files in the *director* folder:

director.html

This is a really useful file when starting out. It's a page full of every MD 'command' and its attributes. It also contains a myriad of useful variables. You can filter the list to narrow down what you're looking for. This file is updated each time new functionality is added to the MD. You may get an Internet Explorer message that it has blocked active content, which will prevent it from working. Right click on the message to allow blocked content, to see the full listing.

director & dirschema XSL stylesheets and director cascading stylesheet

These files provide the formatting information for the XML and HTML files in this folder.

director & dirobjdb XML schema files

These two files provide the naming conventions, look-up functionality and content which is available to the mission designer in their chosen XML editor (if the editor supports the use of schema files). The dirobjdb.xml file gives the designer full access to the 'typename' of all X3 game objects. This speeds up object-related coding immeasurably.

XML EDITING TOOLS

There are a number of free and commercial XML editors available on the internet. In this guide Visual Studio 2005 Web Developer Express edition is used as it fully supports the XML schema used by MD, but unfortunately does not have a tree-view. [This is currently free for download.](#)

Whichever is used will need to have full support for XML schemas (not DTDs) and stylesheets as well as basic XML editing, and should preferably offer syntax highlighting, auto-completion, tree views and so on too. In order to ensure that all developers have the optimum environment for developing their missions, we recommend and prefer that Visual Studio 2005 Web Developer Express edition is used. If all developers are using the same tool, it also helps to reduce support overheads for other less suitable tools, so more time can be spent on mission development.

CREATING A NEW MISSION DIRECTOR MISSION FILE

In this section we will look at how to start the whole process by creating a new MD mission file and the basic steps in producing mission content with XML code. There will be a description of the key elements of the mission file. We will also start to build up a small example mission file as we progress.

Open the *template.xml* file located in the *samples* folder within your *director* folder. You can start creating your mission straight away using this file. Remember to save it fairly early on with a new name and ensure that the saved file remains **in the *director* folder**.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet href="director.xsl" type="text/xsl" ?>
<director name="test" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="director.xsd">
```

Figure 2 Mission Director template

File creation troubleshooting

If you're trying to edit a mission file not located in the *director* folder, then you will find that none of the schema look-ups will work, as illustrated in Figure 3. There are two ways around this. Firstly, you can move the file you are editing into the *director* folder or secondly, you can copy the schema and related files from the *director* folder into your working directory.

A good practice before making a substantial set of changes to your mission is to back it up to another folder, so if things don't work out, you can go back to a previous version.

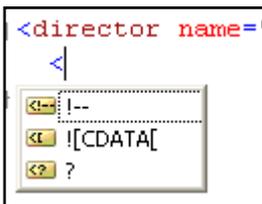


Figure 3 No schema data

If, as in Figure 4, *director.xsd* has an error highlight, this is a further indication that the file has no access to the schema and stylesheet data in the *director* folder. You should move or save your file into there to make use of the MD files. Alternatively you could copy your Director files to your working folder, but you'll need to remember to update both sets as new schema definitions become available.

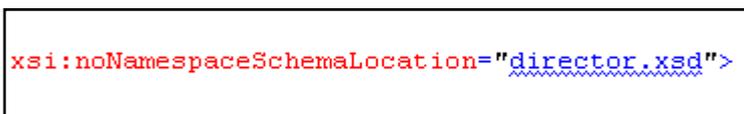


Figure 4 director.xsd – schema not located

XML FORMATTING REQUIREMENTS

Notwithstanding the functionality of the XML editor you are using, there are certain rules in using XML which need to be adhered to, if your mission is to load successfully into the game.

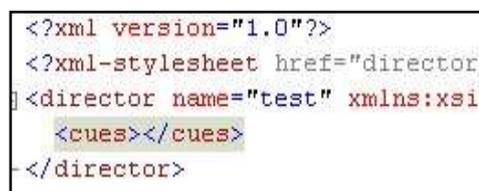
Using indents with tags is important to show the hierarchy of functions of each tag.

```
<parent_node>
  <first_subnode>
    <second_subnode>
      <nested_function/>
    </second_subnode>
  </first_subnode>
</parent_node>
```

It is important to note at this stage that all XML tags need to be closed. This is achieved in one of two ways. Firstly if a tag has no dependent child elements it can be closed at the end of the line, for example: `<find_sector x="0" y="0" name="kingdomend"/>`. If there are any dependent child elements, then the tag is separated as above. As `<parent_node>`, `<first_subnode>` and `<second_subnode>` have dependent child elements the tags are closed after all of the dependent elements by repeating the tag with a / before the tag name as above, for example `</parent_node>`.

<CUES>

The cues tag tells the XML parser that the sub-node `<cue>` will follow. There may be multiple instances of the `<cue>` tag contained within `<cues>` and also there may be further instances of `<cues>` within these.

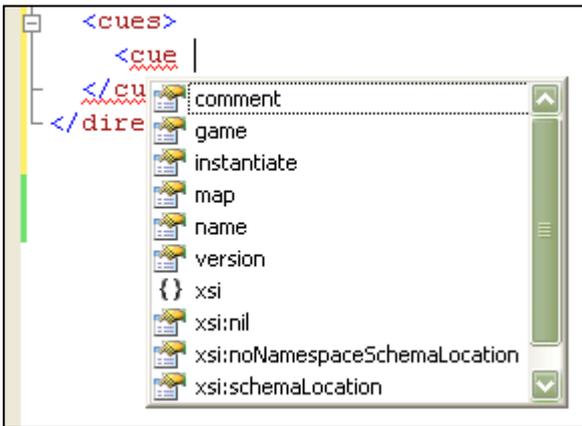


```
<?xml version="1.0"?>
<?xml-stylesheet href="director
<director name="test" xmlns:xsi
  <cues></cues>
</director>
```

Figure 5 `<cues>`

By placing the cursor between the `<cues>` and `</cues>` (see Fig 5) and pressing return, the cursor should move to the right place for the next tag, which is `<cue>`.

<CUE>



The cue tag sets up the environment in which the event will occur, using the attributes seen in Fig 6.

The use of the 'instantiate' attribute can be quite complex. A detailed description of how this works can be found in Appendix 1: Instantiation.

Figure 6 <cue>

The **cue** node has up to four sub-nodes. They are:

Sub-node	Mandatory	Multiple	Notes
<condition>	no	no	Cue triggered automatically if not supplied
<timing>	no	no	Cue actions performed immediately, once, if not supplied
<action>	no	no	Cue events triggered but no actions performed if not supplied
<cues>	no	no	Sub-cues for this cue

Table 1

Table 1 shows the four key sub-nodes that you will work with. **<action>** is the event which occurs a certain **<timing>** after the **<condition>** is met. **<cues>** may occur as a result of the **<action>** with its own sub-cues, conditions, timings and actions.

<CONDITION>

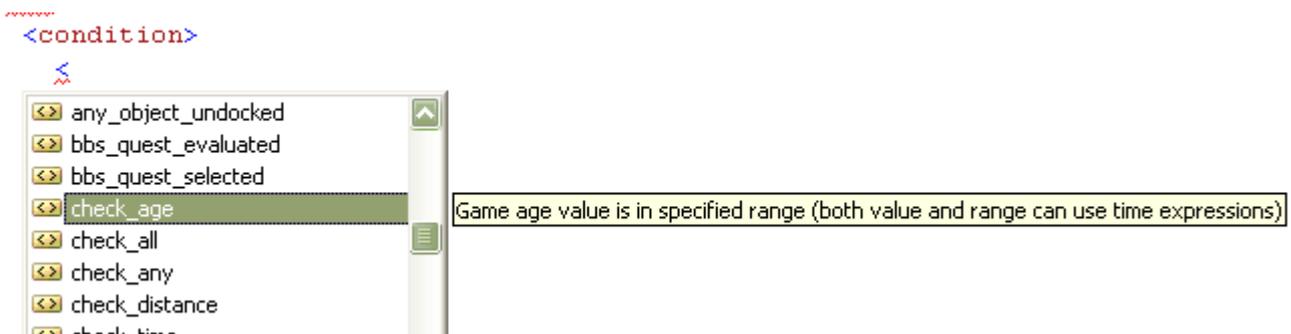


Figure 7 Condition has many available attributes

The Condition component consists of one or more conditions that must be met in order for the Cue to be triggered. These conditions may be based on simple values or ranges, such as a particular in-game time having been reached or the player having a certain amount of money. They may also be based on more complex player information, such as what ships they own, whether the player is in a particular area or near a particular object.

Another important class of conditions are based on notification-type events, such as notification that a particular object has been targeted, attacked or destroyed, or notification that a BBS offer has been accepted or rejected. Conditions can be combined into lists in which all must be met or any one must be met for the Cue to be triggered, and these lists can be nested if necessary in order to model complex trigger requirements.

There can be more than one condition that has to be met for the action to trigger (using <check_all>) or one of a number of conditions could be met to trigger the action (using <check_any>). **It is important to note that when using <check_all>, the conditions are checked in sequence. If the first condition fails, then no others will be checked. So if you are checking events and values in the same cue, always put the events first as they are only usually checked once as they occur, whereas values are checked each time the file is read.** The condition of any cue may be dependent on the status of another cue. For example, once the 'parent' cue has been completed, the condition for the current cue is met and the action is triggered (using <cue_is_complete cue="parent"/>). For an initial cue's condition something as simple as player time (see Figure 7) may be used.

Whichever condition is selected, once met and following the timing/counter criteria outlined in the next section, the action(s) associated with the cue will be triggered.

Here a minimum of 5 seconds and maximum of 7 seconds of game time must expire for the condition to be met.

```
<condition>  
  <check_age value="{player.age}" min="5s" max="7s" />  
</condition>
```

When referred to in a time value, the abbreviations s, m and h are used for seconds, minutes and hours.

VARIABLES

Looking at the example above, you'll notice {player.age} being used in the value="" attribute. This is a 'variable', a piece of information taken from the game and used in the code. There are a number of variables, for which the subject is implicit, for example {player.age}, {player.ship} and {player.money}.

{player.sector.race.name} can be used when you need the race name of the owner of the player's current sector, whichever that may be.

Other variables are similar to those we've just seen, but there is an additional element in which there is a user-defined element showing what the subject of the variable is; such subjects are separated by the @ sign. For example:

{object.pilot@*object*} – if you create a ship and want to get the name of its pilot, you substitute the *object* after the @ for the object name, i.e. {object.pilot@myship} . This is useful in such things as <incoming_message author="{object.pilot@myship}" text="test"/>. One important aspect of variable use in the Mission Director is the creation and use of 'global' and 'local' variables. The creation of a global variable means that the variable should be available to all running xml missions. Making a variable local will mean that as long as it's properly referenced, it will only be available in that mission.

```
<cue name="createopponent">          <- The cue we're creating a ship in
.....
<create_ship name="opponent"         <- This creates the object as a global
<create_ship name="this.opponent"   <- This creates it as a local variable
```

The 'this.' suffix will only be used in the cue that the object and variable is created. If you need to use this variable in a subsequent cue it must be referenced using the cuename of the cue, in which it was created. In the case of our example above, when we need to use the object in another cue, it might appear like this...

```
<incoming_message author={object.pilot@createopponent.opponent}
text="{1081,16}"/>
```

Here we've used the cuename 'createopponent' separated by a full stop/period and then the object name.

Advanced Information on variables:

For the variables representing instance, index and value information, cue names are used. For plot missions and other missions where a single instance is used this is normally sufficient. However, for missions where multiple instances may be running at once (e.g. BBS missions) it is sometimes useful to be able to identify specific instances. When identifying cue instances, the Mission Director will first try the current cue instance, then the parent cue instance, and so on up to the top level of the cue structure. It will also try back down other "branches" of the cue tree structure at each stage.

If it doesn't find a cue of the correct name then it will start looking down all cue structures but may not always find the correct instance (in fact for instantiated cues it will always find the master instance). You can shortcut this process and prevent the latter part of it by using "this" to refer to the current cue instance rather than the cue name, and "parent" to refer to the parent cue instance.

<TIMING>

The Timing component determines exactly how long after the Condition is met the Cue will actually be triggered and its actions performed. The Timing component also determines how often the actions will be performed. The Timing component can cause the actions to be performed a fixed or random number of times, with multiple timings being distributed either randomly or at intervals (again, fixed or random). There can only be one timing node and it is also not mandatory, meaning that the action will occur once and immediately upon all conditions being met.

Sub-node**Notes**

<code><count></code>	Number of times the actions will be performed
<code><time></code>	Time range over which the actions will take place
<code><interval></code>	Interval specification within time range
<code><params></code>	Parameters specified in a library cue

Table 2

Figure 8 shows that between 5 and 10 seconds after the condition is met, the action will be carried out.

```
<timing>
  <time min="5s" max="10s" />
</timing>
```

Figure 8 <timing> example

Figure 9 shows that with a count of 50-100 selected, one of 5 randomising profiles can be chosen, in this case 'increasing' meaning that the number of iterations are likely to be higher (nearer 100 than 50).

```
<timing>
  <count min="50" max="100" profile="" />
</timing>
cue>
es>
ctor>
```

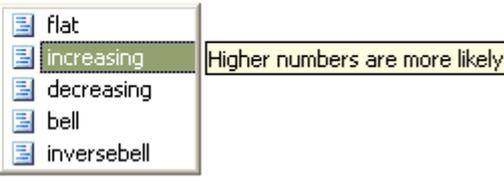


Figure 9 profile attributes

Figure 10 shows an interval of 5s, meaning the action will be repeated at 5 second intervals.

```
<timing>
  <time min="5s" max="60s" />
  <interval min="5s" />
</timing>
```

Figure 10 interval example

UNITS OF TIME AND DISTANCE

Standard units of time and distance are converted into internal values by the MD.

<i>distancem</i>	100m	Converts from metres to internal units
<i>distancekm</i>	{value@this.xpos}km	Converts from kilometres to internal units
<i>timed</i>	5d	Converts from days to milliseconds
<i>timeh</i>	12h	Converts from hours to milliseconds
<i>timem</i>	{value@this.delay}m	Converts from minutes to milliseconds
<i>times</i>	(({value@this.delay}*10)s	Converts from seconds to milliseconds
<i>m:s.ms</i>	1:30.500	Converts from minutes/seconds/milliseconds to milliseconds
<i>h:m:s.ms</i>	1:30:0.0	Converts from hours/minutes/seconds/milliseconds to milliseconds

For time conversions involving : symbols it is advisable to avoid leading zeros in any element, so the milliseconds (ms) should be treated as a number not a decimal fraction! Note that days cannot be specified in this format. The maximum time period that can be used in all cases except the [{player.age}](#) value is 24 days. [{player.age}](#) can be longer time periods but milliseconds are ignored in this case.

EXPRESSIONS

When using variables and working with values, it is possible to use basic mathematical operators in expressions. For example, if you want to check if a sum of money is at least 1000Cr more than the player has you could use the following:

```
<check_value value="{this.cashsum}" min="{player.money}+1000"/>
```

The following expression operators can be used in any attribute whose format is "number" (as opposed to "integer"):

Operator	Example	Description
x+y	{value@this.mylocalvalue1}+{value@this.mylocalvalue2}+2	Add
x-y	{value@this.mylocalvalue1}-{value@this.mylocalvalue2}	Subtract
x*y	{counter@myloop}*10	Multiply
x/y	100/{value@this.mylocalvalue}	Divide (note: division by 0 results in 0)
(x)	(100+{value@this.mylocalvalue})/10	Evaluate bracketed expression first

Operator precedence is () followed by * / and then + -. Operators at the same precedence level are parsed left to right.

<ACTION>

The Action component determines what happens when the Condition is met and the Timing component has determined that the Cue should finally *do* something. Note that the Action component may actually be performed more than once if the Timing component defines multiple timings. A Cue may consist of one single action, such as adding some money to the player's account, sending the player a message, or creating a ship or group of ships, or it may consist of a list of actions. A list of actions may all be performed in sequence, or you can specify that just one randomly selected action in the list should be performed.

```
<action>
```

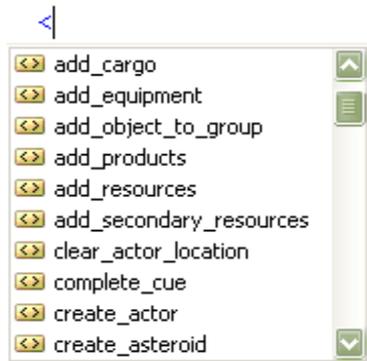


Figure 11 Some <action> sub-nodes

As with Condition lists, Action lists can be nested allowing relatively complex sequences or alternatives to be modelled. An important thing to note, however, is that an Action list is **not** a program. Normal programming structures such as loops and conditionals do not exist in the sense that a programmer might expect. In complex cases the Timing and Condition components should be used to handle loops and conditionals respectively, but for some simple cases there are alternatives which can reduce the complexity of the cue structure. The range of actions that can be performed will grow as new

requirements are found, but the key is that these actions are centrally coded and maintained, minimising repetition and minimising the maintenance overhead.

There should normally be an action node in each cue and there can only be one per cue, so if multiple actions are required these must be represented in sub-nodes, using either <do_all> to perform all of the actions contained within that particular sub-node, or <do_any> to perform one of the listed actions. If neither of those two sub-nodes is used then only one single action will be possible. As alluded to above <do_all> and <do_any> can be nested within each other to provide a wider range of action possibilities.

This is actually also true of <check_all> and <check_any> in the **<condition>** sub-node. This will be explained in greater detail later. Also available is <do_if>, which completes the enclosed action(s) only if one provided value matches another. <do_choose> is a slightly more sophisticated version of the <do_if> that allows you to perform multiple checks and also have an "otherwise"

option which is executed if none of the checks are successful. Sub-nodes of <do_choose> are <do_when> and <do_otherwise>.

In the example below, using the <do_any> sub-node, we can create a bank of incoming messages, for example. When the cue's condition is met, one of the three messages will be randomly selected and sent to the player's logbook. In the particular instance of the <incoming_message> action, the author could be a text string, or more likely a text id. Text is, of course, the message body itself, which can be either free-flow or a text referenced from the game's text database.

```

<action>
  <do_any>
    <incoming_message author="Shipboard Computer" text="Hello World"/>
    <incoming_message author="{1323,119}" text="Welcome to MD"/>
    <incoming_message author="{1323,119}" text="{1278,305}"/>
  </do_any>
</action>

```

Once you have completed the **<action>** node you can insert more dependent **<cues>** or close that particular **</cue>**.

EXAMPLE 1

Here's roughly what you can do so far:

```

<?xml version="1.0"?>
<?xml-stylesheet href="director.xml" type="text/xsl" ?>
<director name="test" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="director.xsd">
  <cues>
    <cue name="evaluator" game="all" version="1" comment="this is only so we
      know what's happening in this cue">
      <condition>
        <check_age value="{player.age}" min="5s" max="7s" />
      </condition>
      <timing>
        <time min="5s" max="10s"/>
      </timing>
      <action>
        <do_any>
          <incoming_message author="Shipboard Computer" text="Hello World"/>
          <incoming_message author="{1323,119}" text="Welcome to MD"/>
          <incoming_message author="{1323,119}" text="{1278,305}"/>
        </do_any>
      </action>
    </cue>
  </cues>
</director>

```

TESTING YOUR FIRST MISSION FILE

Once you've saved your first mission file, check first that it's located in the game's *director* folder. Once again, make sure that none of the cues in your mission are the same as those in existing XML files in the *director* folder.

The next step is to start the game. Start a new game in a custom universe and change your name to Thereshallbewings. If you've set the condition and timing as it is in the example above, then you should very quickly receive one of the three messages or an action of your own should have been triggered.

In order to make testing and debugging easier for the mission developer there is a file which may assist in diagnosing errors in your code or the game.

This is *director.dmp* – this is located in your main game directory and provides a millisecond by millisecond account of what your mission is doing. This includes all the good things as well as any errors.

There will be more on mission testing later.

'IN SPACE' MISSION-WRITING TUTORIAL

In this section you will be guided through the creation of a mission, step by step, with a detailed explanation of each key element of the file. The mission is based on an old X-Tension favourite and is called Docking Race Bet. While this is not the ideal place for a tutorial on how to use VS2005, there are a couple of handy tips you should know before you begin.

- If you lose a look-up list for an attribute, you can get it back by pressing Ctrl-Space.
- The formatting of the file generally looks after itself.
- You can take a better look at your script's cue hierarchy by using the -/+ markers on the left of the page to collapse and expand cues and their elements.
- Hovering your mouse over an entry underlined in blue or red will give you a reason for the problem.
- Hovering your mouse over any element of the code gives further information about it.
- When a list of possible entries is offered to you, highlighting one of these entries will normally give you a description of its meaning.

The first key element of the file is the header, as illustrated in Figure 2, above.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

This line defines the encoding system that will be used to format the file.

```
<?xml-stylesheet href="director.xsl" type="text/xsl" ?>
```

This line defines the stylesheet that XML will use to format the file.

```
<director name="template" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="director.xsd">
```

This long line tells the file where to look to find the schema file, which it uses to define all of the commands and lookups that you will use. Note that the name attribute of the <director> tag is not used in the game, but if the XML file is opened in a browser then the value it contains will be shown at the top of the page and in the page title. This name is for the file as a whole, not for the individual cues within it.

The next couple of lines are fairly self-explanatory and for officially-developed missions the fields are mandatory. The **<documentation>** node tells testers and developers alike who developed the mission and how to contact them for feedback/comments, as well as providing version and specification information about the mission.

```
<documentation>
  <author name="Fred Bloggs" alias="Froggs" contact="froggs@egosoft.com"/>
  <content name="Docking Race Bet" description="player is asked by a fighter
    pilot to race him to the local trading station"/>
  <version number="1.0" date="2006-06-15" status="beta test"/>
</documentation>
```

In these missions we are using an internal mission serial number to prefix some cues. This is by no means essential, but doing so will ensure that there are no conflicts with other XML mission files that may use an otherwise identical cue name. It is absolutely essential that unique cue names are used at the top level of the structure otherwise cues in one file could overwrite those in another - at lower levels sub-cues must be uniquely named within their parent rather than globally, but mission developers should try to get into good habits!

<cues>

The next node in all cases will be **<cues>**. As illustrated earlier, once you have typed <cues>, its closing tag </cues> should appear. Once you have split the tag by pressing return your cursor will automatically be placed for the next sub-node. Before continuing, it's worth reiterating that inside this <cues> node there can be any number of nested <cue> and <cues> nodes. For now though, let's concentrate on where we are.

<cue>

The first cue that we define in any mission is called the top-level cue; that is, all other <cue> and <cues> nodes are subordinate to or stem from the condition, timing and action carried out in this cue. If you like, you can see this as the initial trigger that sets the whole mission in motion.

```
<cues>
  <cue name="S01M25S00prepare">
```

As mentioned earlier, it's important to have a unique cue name here. There are other <cue> attributes that can be specified at the beginning of a mission.

game="" will determine in which game mode(s) the mission will be available in, for example, all, plot, noplot, custom (If not specified, the default is 'all').

`map=""` specifies a pre-defined map that the mission will use.

`instantiate=""` 'static' creates a working copy of instantiated cue(s) independent of the original(s). This applies to any cue, not just at the top-level. See Appendix 1 for more.

`library=""` sets the cue as 'library' only, meaning that all of the conditions, timing and actions therein will only be carried out if later referenced using `ref=""`.

`ref=""` is used to refer to a library cue.

`comment=""` is used to provide commentary on the function of the cue. `comment=""` is available for all nodes and should be used liberally to inform testers and developers of the purpose of that node. This helps immeasurably when debugging and troubleshooting.

A full explanation of the use of `library=""` and `ref=""` can be found in *Appendix 2: Library Cues*

Next come the three crucial elements for each cue:

<condition>

In our mission there are a number of conditions that must be met for it to start.

```

1 <condition>
2   <check_all>
3     <object_changed_sector comment="check this first, other conditions are not
4       checked if this is false already"/>
5     <match_object class="fighter" comment="M3, M4 or M5"/>
6     <check_value value="{player.age}" min="1m"/>
7     <!--
8     <check_age min="10m" chance="(72000/{player.time})+1" comment="this should
9       reduce the chance of the mission being offered from 100% after 10 minutes
10      down to 1% after 20 hours"/>
11     -->
12     <object_has_equipment comment="if we have a docking computer, we'll race in
13       space">
14       <ware typename="SS_WARE_TECH241" min="1"/>
15     </object_has_equipment>
16     <check_value value="{player.money}" min="{reward.money@veryeasy.XXXT}+500"/>
17     <check_value value="{player.sector.race}" exact="{lookup.race@argon}"/>
18   </check_all>
19 </condition>

```

Let's deal with this <condition> node line by line.

1 & 15: The open and close tags for Condition, that's fairly easy.

2 & 14: <check_all> allows us to place a number of conditions, all of which **must** be met.

3: The first condition to be checked is <object_changed_sector> which is an event condition and is met when the object arrives in a new sector or docks. If no object is specified, it refers to the playership.

4: This line is checking if the player's ship matches the class 'fighter', M3, M4 or M5.

5: Here we check that the age of the player {player.age} is at least one minute. This is just a debug/testing value. Player age refers to the elapsed game time.

6&8: `<!--` and `-->` can be placed either side of code that you do not wish to be parsed, this 'commenting out' is a useful temporary way of disabling parts of your code. It goes without saying it can also be used for commentary pertaining to sections of your mission code.

7: This alternative condition has been 'commented out', as the one above is being used, for the moment, as a debug value, to allow for easier testing. In the game proper the condition should function as described in the comment field.

9-11: Here we're checking that the player has a docking computer. The `<ware>` subnode must be used to specify the typename of the ware you want to check for, in this case it's `SS_WARE_TECH241`, which is in the lookup list that appears. As mentioned earlier, it's good practice to provide commentary on the steps you take. One important thing to note in this sub-node is that when no object is specified using `object=""`, it will refer to the playership by default. This is very useful to bear in mind. You can have more than one `<ware>` sub-node allowing you to specify several items that the object must have.

12: This line checks that the money in the player's account {player.money} is at least the same as the reward money for the mission+500Cr. Just in case the player loses the bet. The pre-balanced rewards for each type of mission and difficulty can be found in the *director.html* schema file.

13: The final check, although complicated at first sight, is comparing two variables, the first {player.sector.race} represents the internal ID code used by the game for the race of the sector in which the player is located. If this value then exactly matches the value for {lookup.race@argon} which provides the internal ID code for the Argon race the check is successful. So simply put, this checks that the sector the player is in is Argon.

So there's quite a few things which need to be true for the `<condition>` node to be met. Do remember that `<check_all>` means **all** conditions have to be met.

TIP: If you need to have a `<condition>` node where some conditions must be met and perhaps one of a number are met, you can nest a `<check_any>` within the `<check_all>`.

On the other hand, if you have a number of blocks of conditions which must be met, but you want one of those blocks to be selected randomly, then the nesting would be the opposite way round i.e. `<check_all>` nodes nested within a `<check_any>` node.

<timing>

In this top-level cue the <timing> node is fairly uncomplicated:

```
<timing>
  <time min="3s" max="5s"/>
</timing>
```

This timing means that at some time between 3 and 5 seconds after all the conditions of the cue have been met, the action will be triggered.

<action>

As with the <condition> node, there's a fair bit going in the <action> node, so we'll break it down again.

```
1  <action>
2    <do_all>
3      <find_gate nearest="1" name="this.gate">
4        <distance max="10km"/>
5      </find_gate>
6      <find_station name="this.finish" class="trade" dockingallowed="1"
7        findobject="{player.ship}" max="1" typename="SS_DOCK_A_TRADE"/>
8      <set_value name="this.reward"
9        exact="({reward.money@veryeasy.XXXT}/100)*100" comment="extra /100 *100
        to fake a rounded value ;)/>
10   </do_all>
11  </action>
```

1&9: The opening and closing tags of the <action> node.

2&8: The <do_all> sub-node means that all actions contained within will be performed

3-5: The <find_gate> sub-node will in this case look for the nearest gate and also, as the <distance> sub-node suggests it has to be within a maximum radius of 10 km. Once found, such an object will be given the name “this.gate”. The name here is important, if it were just “gate” then that name would become a global variable, available to the whole game and all loaded MD files. As it is, we only want it to be local and therefore only available to the cue in which it is declared. To do this we prefix the chosen name with ‘this.’

6: Next we want to find a station, which will be known as “this.finish” and must be of class “trade”, ie a Trading Dock. Specifically we want to find only one of type SS_DOCK_A_TRADE, the Federal Argon Trading Dock. Additionally, it needs to be a station at which the object {player.ship} is able to dock.

7: The <set_value> sub-node ascribes a value or range of values to a name, in this case ‘this.reward’ which can then be used as a variable. As with the other names we’ve encountered so far, this is a local variable. The value ascribed to this name is defined in the calculation $((\text{reward.money@veryeasy.XXXT})/100)*100$. When performing calculations, you can only use whole numbers. If you use brackets, they will be evaluated

first. If there are nested brackets (not including variable braces { }) they will be evaluated inner one first. So in our example, it's the prebalanced reward for a very easy XXXT mission. The result of that is then divided by 100. The result of that calculation will then be multiplied by 100. As the comment suggests the /100*100 attempts to give a rounded value to the original value.

That's the first cue dealt with. In that cue we set the scene in terms of what conditions need to be met to begin the mission, then a short time after those are met, we look for a gate and a station, which will be the finish point of the race. Also we have set the amount of money the player will win if successful,.

In the next cue, which is a sub-cue of the first, there is a little housework to be done before the action starts. The aim of this cue is to check that the items we searched for in the previous cue, the gate and the station, have been found. We'll look at what happens if they're not found next. There's also a good example of nested conditions in this cue.

First, what makes a <cue> a sub-cue? It was suggested earlier in this document that cues can be nested within cues. In order for this to happen we need to introduce another <cues> node before closing our existing cue. This <cues> should directly follow the </action> tag of the cue it is following, for example:

```
</action>
<cues>
  <cue name="blah">
```

In this way <cues> node introduces at least one more <cue>. In reality there can be as many cues as necessary at this level, with sub-cues of any of those also a possibility.

<condition>

So let's take a look at the second cue, looking at the conditions first...

```
1  <cues>
2    <cue name="reset">
3      <condition>
4        <check_all>
5          <cue_is_complete cue="S01M25S00prepare"/>
6          <check_any>
7            <object_exists negate="1" object="S01M25S00prepare.gate"/>
8            <object_exists negate="1" object="S01M25S00prepare.finish"/>
9          </check_any>
10         </check_all>
11      </condition>
```

1: As just mentioned, we're starting a sub-cue with <cues>, so the next tag must be <cue>.

2: As promised... We're going to name this cue 'reset' for a very good reason, which will become clear as we progress.

3&11: The open and close tags of the <condition> node.

4&10: <check_all> will check that all of the conditions contained within are met.

5: The first condition which must be met is that the cue named “`S01M25S00prepare`”, which you’ll recall was the name of our top-level cue, has to have been completed, i.e. all conditions met and all actions performed.

6&9: The `<check_any>` node nested within the `<check_all>` will be checked but either of the conditions within must be met for the actions to be triggered. Consider this nested condition to equate roughly to ‘and if either’.

7&8: We need to check if either the gate or the station, that were the subject of ‘find’ commands in the parent cue were actually located. It essentially asks if either the `.gate` or the `.finish` object does **not** exist. If either of these evaluate as true then the conditions as a whole are not met. You may be wondering why the objects have the `S01M25S00prepare` prefix, where previously they had `this`; `this.blah` is a local variable within the cue it’s declared. If we need to refer to it in a subordinate cue we substitute ‘`this`’ for the name of the cue, in which it was declared. If we refer to a local variable in the parent cue, it’s also possible to use `parent.blah`.

<timing> and **<action>**

Now that we’ve run the check to see if the top-level cue is complete and if either the gate or the station were not found there needs to be some action taken. So what are the consequences of there being either no gate or station. It’s clear that if neither are found then our mission won’t work too well.

```
<timing>
  <time min="1s"/>
</timing>
```

Again there’s a fairly straightforward timing node. One second after the conditions have been met, the action below will be triggered.

```
<action>
  <reset_cue cue="S01M25S00prepare"/>
</action>
</cue>
```

Not a huge action node this time, but important given the conditions we had just now. So now it’s established that the mission is effectively cancelled because the objects couldn’t be found - the result is that no further attempt to run the mission will be made until the player changes sector again - only **then** will the gate and station be searched for once more. That means that the check for the gate and the station will begin from scratch and will continue until both have been found.

The `</cue>` tag after the `<action>` node means that this is the end of the line for this cue, there are no more sub-cues. The next cue looks a little tricky as it is very big, but we will break it down into manageable chunks.

```
1<cue name="selectOpponent" comment="selecting a random typename to match the
  playership">
2  <condition>
  <cue_is_complete cue="S01M25S00prepare"/>
3 </condition>
```

1: This is the cue name, which with the provided comment suggests the purpose of this large cue.

2-3: The condition of this cue is simply that the parent cue (`s01M25S00prepare`) has completed all of its conditions and actions.

The `<action>` node of this cue is a whopper, so brace yourself for some heavy stuff.

```

1  <action>
2  <do_choose>
3  <do_when value="{player.ship.class}" exact="{lookup.class@m3}">
4  <set_value name="selectOpponent.typename"
   exact="{lookup.type@{random.type@SS_SH_A_M3|SS_SH_A_M3_1|SS_SH_A_M3_2|SS_SH_
   A_M3_3}}"/>
5  </do_when>
6  <do_when value="{player.ship.class}" exact="{lookup.class@m4}">
7  <set_value name="selectOpponent.typename"
   exact="{lookup.type@{random.type@SS_SH_A_M4|SS_SH_A_M4_1|SS_SH_A_M4_2|SS_SH_
   A_M4_3}}"/>
8  </do_when>
9  <do_otherwise comment="not M3 and not M4 so I'm expecting M5">
10 <set_value name="selectOpponent.typename"
   exact="{lookup.type@{random.type@SS_SH_A_M5|SS_SH_A_M5_1|SS_SH_A_M5_2|SS_SH_
   A_M5_3}}"/>
11 </do_otherwise>
12 </do_choose>
13 </action>
14</cue>

```

1&13: The `<action>` tags here are the easy bit.

2&12: All of the actions in this cue are governed by the `<do_choose>` sub-node, which allows you set additional ‘conditions’ for the execution of the actions. It also has a loose sense of ‘if a=b then do c, otherwise do d’. We’ll see how that works.

3&5: The `<do_when>` sub-cue is framed here, the ‘condition’ here is that when the internal ID for the class of the playership is the same as the look-up value for M3 class.

4: Once the player ship is matched to the M3 class, the typename of the opponent is assigned a lookup value (an internal ID). Here, we can use the pipe symbol | in the `random.type` variable to mean ‘or’, resulting in the typename of the opponent being any one of the four Nova variants selected randomly.

6&8: Similar to 3&5, this time we check if the internal ID of the playership’s class matches the lookup value of an M4.

7: This is almost identical to **4**, but on this occasion if the playership is an M4, one of the four Argon M4 Buster variants will be selected as the opponent.

9&11: Instead of `<do_when>` we have `<do_otherwise>`, the failsafe clause in case the other ‘conditions’ aren’t met. Here, if the player ship isn’t M3 or M4 it must be M5 (we’ve already established right at the top that the mission will only work if the playership is a fighter).

10: One of the four variants of the Argon Discoverer (M5) can be assigned as the opponent. Note: In all of the <set_value> sub-nodes the internal ID (look-up value) of the typename for the selected variant is given the name “selectOpponent.typename”. This variable can then be used when creating the opponent in the following cue.

14: This is the cue being closed; there are no further sub-cues of ‘selectOpponent’.

The next cue is on the same level and like its predecessors it is a sub cue of the top-level cue S01M25S00prepare.

```
<cue name="createopponent">
```

As before, we’ll deal with the cue by node, first of all, looking at the <condition> node. Probably worthy of note at this stage, in case it goes unnoticed, is that there is no timing node. This means that the action(s) will be triggered immediately once the conditions have been met.

```
1 <condition>
2   <check_all>
3     <cue_is_complete cue="selectOpponent"/>
4     <cue_is_complete cue="S01M25S00prepare"/>
5     <object_exists object="S01M25S00prepare.gate"/>
6     <object_exists object="S01M25S00prepare.finish"/>
7   </check_all>
8 </condition>
```

The conditions, which must be met in this cue, are similar to the ones we encountered in the cue before last, called “reset”.

1&8: The condition node tags

2&7: The conditions are encapsulated within a <check_all>, meaning all have to be met.

3: Cue selectOpponent has to have been completed.

4: Cue S01M25S00prepare, our top-level cue, has to have been completed.

5: The gate object we looked for in the top-level cue should have been found and exists.

6: The station object we looked for in the top-level cue should have been found and exists.

So, all being well, we’ve selected an opponent, the preparatory location of a gate and a station were completed and we’ve checked that those two actually exist as game objects, we can now proceed with the business suggested by the cue’s name, creating the ship against which the player will race.

Warning! There’s a lot of action in this next part, so if you need coffee or popcorn, you’d best grab it now. ☺

As we mentioned just before, there's no <timing> node, so it's straight into the action in this cue.

```

1<action>
2  <do_all>
3    <create_ship name="this.opponent" typename="{value@selectOpponent.typename}"
4      racelogic="0" race="argon" class="{player.ship.class}" highlight="1">
5      <position min="5km" max="6km" object="S01M25S00prepare.gate"/>
6      <equipment loadout="default"/>
7      <command command="follow" commandobject="{player.ship}"/>
8      <pilot name="{random.pilot.argon}" race="argon"/>
9    </create_ship>
10   <set_value name="this.tunings"
11     exact="{player.ship.maxspeed}/({object.basespeed@this.opponent}/10)-10"/>
12   <add_equipment object="this.opponent">
13     <ware typename="SS_WARE_TECH213" exact="-100" comment="removing all engine
14       tunings as a workaround"/>
15     <ware typename="SS_WARE_TECH213" min="{value@this.tunings}-1"
16       max="{value@this.tunings}+1" comment="Engine Tunings"/>
17   </add_equipment>
18   <do_if negate="1" value="{object.class@this.opponent}" exact="{lookup.class@m5}"
19     comment="M5 ships can't carry jumpdrives, so don't try to add it">
20     <add_cargo object="this.opponent">
21       <ware typename="SS_WARE_ENERGY" min="15" max="50" comment="Energy Cells"/>
22     </add_cargo>
23     <add_equipment object="this.opponent">
24       <ware typename="SS_WARE_WARPING" exact="1" comment="Jumpdrive"/>
25     </add_equipment>
26   </do_if>
27   <ask_question name="bet" author="{object.pilot@this.opponent}" text="{1081,1}"/>
28 </do_all>
29</action>

```

Easy ones first:

1&24: Our <action> node tags.

2&23: <do_all> sub-node tags meaning that the actions in lines 3-22 will all be done.

3&8: We want to use <create_ship> and its sub-nodes to give us our opponent. Attributes that we can specify on this line include a name, in this case *'this.opponent'*. We can also specify a typename, an internal code specifying which model will be used in the game. In this instance, we're using a value which was specified in the selectOpponent cue. Racelogic is set to 0, meaning that upon creation it won't wander off thinking it's a normal race ship, in this case an Argon one of the same 'class' as the player's ship. This ship, with highlight="1", will appear highlighted underneath the player's assets on the sector map.

4: The ship we're creating will spawn in a position somewhere between 5 and 6km from the gate object that we found in the top-level cue. Note once more that instead of *this.gate*, as it was declared as in that cue, we're referring to it as *S01M25S00prepare.gate* so that the code knows where to look to find where the object was declared in order to use it.

5: Here we want to install some equipment on the opponent's ship, we can specify a general load-out for the ship, in this case 'default', but specific items of equipment can be installed using the <ware> sub-node. There are three different types of 'loadout':

- minimum (one 1MW shield, one set of lasers - usually front turret, no missiles, only basic upgrades, no speed/rudder/cargo upgrades)
- default (max shields with a small chance of something less, max lasers with a small chance of something unusual, some missiles and a variety of equipment)
- maximum (max shields, max weapons with a higher chance of something unusual, max missiles and a variety of equipment similar to 'default')

6: As part of the <create_ship> node we can give our ship a command to carry out as soon as it has been spawned, in this case it should "follow" the player ship (the commandobject).

7: Next we use the <pilot> sub-node to set the pilot's name, in this case we're using a variable to find a randomised Argon name, but a text string or TextID would also suffice. Also here we can set the race of the pilot to Argon. Note that this is the race of the pilot, as distinct from the race of the ship.

9: This sets the value of the variable {this.tunings} to the result of the calculation given in value="". That is the player's maximum speed divided by a tenth of the opponent's base speed minus 10. This number can then be used as a multiplier when adding tunings to the opponent's ship.

10&13: Here we have a separate <add_equipment> sub-node. The aim of this is to add equipment to the 'opponent' using the value we have just set in **12**.

11: This <add_equipment> node will remove 100 engine tunings from the opponent's ship, with a view to adding more in just a second. This should in theory remove any tunings the opponent ship has installed.

12: This <add_equipment> node then adds an amount of engine tunings somewhere between {value@this.tunings}-1 and {value@this.tunings}+1.

14&21: Here we're going to use a <do_if> sub-node. The negate="" is used to check that a condition or action is **not** true; in this case, if the internal ID of the opponent does not match the look-up value for an M5 ship. As the comment suggests, if the ship is an M5 then it can't be fitted with a jumpdrive, so the actions contained in the <do_if> should only be carried out on M3 or M4 ships.

15-17: In the <add_cargo> sub-node we're adding between 15 and 50 energy cells to the opponent's ship.

18-20: In this <add_equipment> sub-node, we're adding a single jumpdrive to the opponent's ship. This equipment isn't part of the race, it's just so the opponent can 'disappear' as required.

22: The final action in this cue is an <ask_question> sub-node. The question is given a name, by which it will be known in subsequent sub-cues. The question will appear as an incoming message of whichever type is specified; so it has an author, in this case the name of the pilot of the *this.opponent* ship we created. Then follows the text of the

question. In this case we are using a TextID, but it could as easily be a text string. Almost all official mission text will be placed in the Text Database, so reference to TextIDs will be more common than text strings. Text used should also be as brief as possible.

Phew! So now we have created our opponent and asked him a question. To get a better idea of what will come next, let's take a quick look at the question:

This is {object.pilot@createopponent.opponent} of the Argon. I've seen you have a fast ship!\nShall we bet for {value@S01M25S00prepare.reward} credits I can dock at the {object.name@S01M25S00prepare.finish}, the large station in the middle of this sector, before you can?

[center][select value='yes']You're on![/select][/center]

[center][select value='no']No thanks![/select][/center]

There are a few variables in there. **{object.pilot@createopponent.opponent}** is the name of the pilot of the ship created as this.opponent; it's now using the cuename rather than 'this.' so that the code can properly locate the variable.

{value@S01M25S00prepare.reward} was declared earlier in the top-level cue with **{object.name@S01M25S00prepare.finish}**, both of which show their cuename origin in the variable. So the object formerly known as this.opponent of the Argon wants to bet that he can beat you to the station we originally called 'this.finish'.

The use of the <ask_question> node requires there to be at least one sub-cue, but normally two – one each for a 'yes' and 'no' answer. Each of those sub-cues will check if the 'yes' or 'no' answer has been selected and then perform a further action based on that selection. So let's now look at what happens in response to *this.opponent's* challenge.

As the answer is going to be a sub-cue of the question's cue the next line of code will be <cues> directly following the <action> node of the parent cue. After that, follows the cue name.

```
</action>
<cues>
  <cue name="noBet">
```

As expected, the condition of this sub-cue will be to check one of the answers of the question posed in the parent cue.

```
<condition>
  <question_answered question="bet" answer="no" />
</condition>
```

Note that the question="" attribute has "bet" in, which is the name we assigned to the question in the previous cue. So this condition is checking if the answer to the question 'bet' was no, if so the cue will perform the following action:

```
<action>
  <do_all>
    <incoming_message author="{object.pilot@createopponent.opponent}"
      text="{1081,9}"/>
    <cancel_cue cue="closetoplayer"/>
  </do_all>
</action>
```

`text="{1081,9}"` is **"I should have known that someone who flies such a rustbucket is not up for a challenge - get that junk out of my sight!"**

This cue is fairly straightforward in comparison to the last cue; the actions in this cue send a message to the player, as displayed above and also cancel the cue that follows it. Note that cancelling a cue also cancels any sub-cues. As it happens, the cue that is cancelled is the cue, which is the 'yes' answer to the challenge. So answering 'no' effectively ends the mission, as one would expect. Before we get to that cue, there is another small matter to attend to.

```
<cues>
  <cue ref="clean up">
    <params>
      <param name="time" value="10s"/>
    </params>
  </cue>
</cues>
```

This very small cue, wrapped neatly in `<cues>` tags, is a sub-cue of 'no bet' and directly follows the `</action>` tag of the previous cue. As we saw earlier the `ref=""` attribute is used to refer to a library cue. So, later in our mission file we will have another library cue called 'clean up'. The purpose of this 'clean up' library cue is to tactfully remove our unwanted opponent from the game once the player has answered 'no'. We'll look at how this is achieved in greater detail once we've reached the 'clean up' cue. The purpose of the `<params>` node and its `<param>` sub-node is to provide a specific value for that library cue to use in this instance. Each time the 'clean up' cue is referenced, differing parameters can be supplied as required. A more detailed explanation of how `<params>` work is available in *Appendix 2: Library Cues*.

The next cue, like 'no bet' is a sub-cue of 'createOpponent' and not the previous one, so in order to make that distinction we must first end the 'no bet' cue using `</cue>...`

```
</cue>
<cue name="closetoplayer">
```

This is the cue that gets cancelled (along with all its sub-cues) if the player answers no. For now we need to be optimistic and hope the answer is yes. As we saw with 'no bet', the condition is simply checking to see if the answer to the question is 'yes'.

```
<condition>
  <question_answered question="bet" answer="yes" />
</condition>
```

Unlike 'no bet' this cue has a `<timing>` node, waiting between 1 and 3 seconds before the actions are triggered:

```
<timing>
  <time min="1s" max="3s"/>
</timing>
```

Next, the action bit:

```

<action>
  <do_all>
1   <incoming_message author="{object.pilot@createopponent.opponent}"
      text="{1081,2}" popup="1"/>
2   <create_object name="this.marker" typename="SS_SPECIAL_MARKER"
      class="special">
      <position min="3km" max="5km" object="{player.ship}"/>
    </create_object>
3   <set_target object="this.marker"/>
4   <set_command object="createopponent.opponent" command="moveposition"
      commandobject="this.marker">
      <position object="this.marker"/>
    </set_command>
  </do_all>
</action>

```

Here we have a bunch of actions all wrapped up in <do_all> tags. From this point we'll just number the action (and condition) lines, assuming you're now comfortable with the formation of the major node types like <cue>, <condition>, <timing>, <action> and <do_all/any>.

1: First up is a message from our opponent, this time it's a popup message, meaning that rather than being received as an incoming message, with the accompanying sound, this message will just 'pop up' on the screen. The message is **"Excellent! I will switch off my friend/foe signal so you can see me as a red dot on the radar. Convenient, huh?! Don't shoot me, I'm not the enemy! Please fly to the marked coordinates. You need to be within 200m of the marker!"**

2: Next we're creating a marker, representing the start point and we'll call it 'this.marker'. Remember: if this object should appear in other cues, it will be called 'closetoplayer.marker'. We scroll down the typename 'lookup' list to find the correct one, and similarly the class is also 'special'. In a sub-node of <create_object> we can specify an approximate position for the marker. It should appear between 3 and 5km away from the player ship.

3: Next we use the <set_target> node to set the player's target as the marker we've just created. Note that because we're still in the same cue, the marker is still called 'this.marker'.

4: Here, using <set_command>, we're telling the opponent ship to move to the marker. Specifying the marker as the commandobject and in the <position> sub-node will ensure the ship gets to the right location.

So, to recap on what's happened in our mission so far, we've identified a station (the destination) and a gate from which (approximately) the race will start. If both are found to exist, then an opponent with speed that roughly matches the player's will be spawned and will issue a challenge to race the player to the destination station. If the player says 'no', the opponent throws a rebuke at the player and disappears and no more is said. If the player says 'yes' a marker, to which the player and opponent fly, is created.

As was suggested earlier, all of the mission elements that flow from having answered 'yes' to the challenge should be sub-cues of that particular cue and if the cue was cancelled as a result of the answer being 'no' then none of them would happen. As we're

still looking on the bright side of life, we can move on to the next sub-cue, in which we'll see what happens if the player develops a yellow streak.

For a change, let's look at the cue as a whole:

```
<cues>
  <cue name="coward">
    <condition>
      <check_all>
        <object_changed_sector/>
      </check_all>
    </condition>
    <timing>
      <time min="1s"/>
    </timing>
    <action>
      <do_all>
        <incoming_message author="{object.pilot@createopponent.opponent}"
                          text="{1081,114}" popup="1"/>
        <destroy_object object="closetoplayer.marker"/>
        <cancel_cue cue="inposition"/>
      </do_all>
    </action>
  </cue>
</cues>
```

In doing so, we can see its structure clearly, the condition, the timing and the action nodes. The **condition** node has a `<check_all>` which isn't strictly needed, as there's only one condition. It helps to have it there in case any other conditions need to be added later. The condition `<object_changed_sector>` refers, if no other object is specified, to the player. So if the player leaves the sector for whatever reason, the action will be triggered after the gap of one second specified in the **timing** node. The **action** node, in this case, contains three actions. First the opponent sends a message to the player "**Coward! I've seen you leaving the sector. The bet is off!**"

Then, using `<destroy_object>`, the marker is removed from the game (note the cue name use in the object name). Finally, the cue 'inposition' is cancelled, along with its sub-cues.

If you're wondering, at this stage, how the mission could run with so many cues being cancelled or reset, you should bear in mind that the game engine is scanning through all the XML mission files every second. The scan filters through the cues from the top-level downwards, checking that each condition has been met. If a condition is not met then it will continue to check the next cue down and so on until the end of the file. This can mean that if you don't cancel a cue once it and its sub-cues are no longer needed their conditions could still continue to be met and actions carried out once you think the mission is over.

The next cue is a sub-cue of 'coward'. It is another short one.

```
<cues>
  <cue ref="clean up" comment="10 seconds from now the opponent will jump to
    another sector where it will be destroyed"/>
</cues>
</cue>      <- this is the 'coward' clue being closed
```

As we had before, a cue with a `ref=""` attribute and this time we're 'calling' the 'clean up' library cue again. The comment provided in the code adequately describes what we're

trying to achieve in this cue. You'll notice that there is an additional `</cue>` at the end. This is because we are ending the 'coward' cue and the next cue will be a new cue at the same level as 'coward', a sub-cue of 'closetoplayer'.

```
<cue name="opponentready" comment="we shouldn't need this cue but the current
Flight AI does">
  <condition>
    <check_all>
1      <cue_is_complete cue="closetoplayer"/>
2      <object_position object="createopponent.opponent" max="200m">
        <position object="parent.marker"/>
      </object_position>
    </check_all>
  </condition>
  <action>
3    <set_command object="createopponent.opponent" command="none"/>
  </action>
</cue>
```

In the `<condition>` node of this cue we are checking:

1: that the actions of the parent cue 'closetoplayer' are complete (even though it's not directly following, this cue is still a sub-cue of it) and

2: that the object 'opponent' is no more than 200m from the object 'marker'. Note here it's referred to as 'parent.marker' meaning that the object 'marker' is declared in the parent cue, it could just as easily be described as 'closetoplayer.marker'. Both would evaluate correctly. So once both of those conditions are met the action will be triggered.

3: is the action that will be triggered. In this instance, we are giving the object 'opponent' the command to stop where he is, using the command 'none'

The following cue, once more is on the same level as the last and a sub-cue of 'closetoplayer'. This time, instead of checking the opponent's position relative to the start 'marker', we're going to check both the player's and the opponent's position relative to the starting position.

```
<cue name="inposition">
  <condition>
    <check_all>
1      <cue_is_complete cue="closetoplayer"/>
2      <object_position object="createopponent.opponent" max="200m">
        <position object="parent.marker"/>
      </object_position>
3      <object_position max="200m">
        <position object="parent.marker"/>
      </object_position>
    </check_all>
  </condition>
  <action>
    <do_all>
4      <incoming_message author="{object.pilot@createopponent.opponent}"
        text="{1081,8}" popup="1"/>
5      <set_command object="createopponent.opponent" command="none"/>
6      <set_relation object="createopponent.opponent">
        <relation relation="enemy" relationobject="{player.ship}"/>
      </set_relation>
7      <set_target object="S01M25S00prepare.finish"/>
    </do_all>
  </action>
```

Just like the preceding cue, we're checking

- 1: that all of the actions in 'closetoplayer' have been completed.
- 2: We again check the opponent's position relative to the start marker; within 200m.
- 3: We are checking the player's position relative to the marker, no more than 200m, just like the opponent. Once those conditions are met, all of the following actions are triggered:
- 4: The player receives a pop-up message from the opponent "**Good, you are in position - starting the countdown...**"
- 5: We tell the opponent to stop where he is, using the command 'none'.
- 6: We then set the relation of the opponent, using the <set_relation> sub-node, to enemy in relation to the object {player.ship}. This is consistent with the message the player received in the parent cue regarding the opponent switching off his friend/foe signal.
- 7: The final action of this cue is to set the player's target to the station we set right at the beginning as the 'finish'.

You're possibly wondering why the opponent's position was checked twice in separate cues. In the first one, we're getting the opponent ship to hold station if it's within 200m of the marker. It may be possible that the opponent's ship may arrive in position before the player. In this case it will simply stop and wait. If, however, the opponent arrives inside

that 200m area at about the same time as the player then a different set of actions will be triggered.

The message passed to the player above gives a clue to what we'll do in the next cue, which is a sub-cue of 'inposition'. We're going to perform a countdown:

```
<cues>
  <cue name="countdown" comment="count down from 4 to 1">
    <condition>
1     <cue_is_complete cue="inposition"/>
    </condition>
    <timing>
2     <count exact="4"/>
3     <time min="2s"/>
4     <interval exact="1s"/>
    </timing>
    <action>
      <do_all>
5       <set_value name="this.counter" exact="5-{index@this}" comment="can't
        compute the number in the text attribute, so make it a value first"/>
6       <play_subtitles author="{object.pilot@createopponent.opponent}"
        text="{value@this.counter}!"/>
7       <play_sound soundid="924"/>
      </do_all>
    </action>
  </cue>
```

While it may look a bit tricky, this sub-cue is fairly straightforward.

1: The condition here is the easiest bit by far, we're just checking if the actions of the parent cue, 'inposition', have been completed. One that condition is met we have the <timing> node to consider. This is the first time we have encountered <timing> node containing, <count>, <time> and <interval>.

Let's look at these elements individually:

2: <count exact="4"/> means that the actions contained in the <action> node will be carried out exactly four times.

3: <time min="2s"/> means, as before, that the actions (the first iteration, in this case) will be carried out at least two seconds after the condition is met.

4: <interval exact="1s"/> means that each iteration of all the actions being completed will happen at exactly one second intervals.

When it now comes to the actions, it's important to remember that they're being repeated at one second intervals, especially so when looking at the <set_value>. Again for the sake of clarity, we'll take a look at the actions individually.

5: Here we're giving the value name 'this.counter' a value of 5 minus {index@this}. What this means is that for each time the actions are read (according to the <count> sub-node, four times), {index@this} is replaced by a number starting at one and rising each time. In this way, in each iteration, the value of 'this.counter' decreases from 4 to 3, to 2 and to 1 (i.e. 5-1, 5-2, 5-3 and 5-4). As the comment suggests, we can't put this calculation in the text field of the next action, so we ascribe it a value here and then use the value in the text field.

6: Here we are displaying a subtitle on the screen displaying the value of 'this.counter', described above. In each iteration of the actions the subtitle should read "4", "3", "2" and then "1".

7: Accompanying each iteration and countdown number appearing on the screen, the game will play sound ID '924' (Alert). In this way the player gets an audible countdown signal as well as the visual subtitled countdown.

That then is the end of that cue and there are no sub-cues of that. The next cue is another sub-cue of 'inposition', where the player's position relative to the marker was checked. In this cue, if the player tries to get a head-start then it's all over.

```
<cue name="headstart">
  <condition>
    <check_all>
1      <cue_timer cue="inposition" min="3s" max="6s"/>
2      <object_position min="220m">
        <position object="closetoplayer.marker"/>
      </object_position>
    </check_all>
  </condition>
  <action>
    <do_all>
3      <incoming_message author="{object.pilot@createopponent.opponent}"
        text="{1081,16}" popup="1"/>
4      <destroy_object object="closetoplayer.marker"/>
5      <cancel_cue cue="countdown"/>
6      <cancel_cue cue="coward"/>
7      <cancel_cue cue="go"/>
    </do_all>
  </action>
```

1: The <cue_timer> condition evaluates the elapsed time, for which a cue has been active. In this case we are checking that the 'inposition' cut has been running for between 3 and 6 seconds.

2: Next we check to see if the player's position is at least 220m away from the marker. Remember that if no object position is specified in the <object_position> sub-node then the game will assume it is the playership object.

So here's what happens if we try to get a run on the opponent.

3: For trying to cheat the opponent out of his money the player gets the following message, "**You honourless cheater! I've seen exactly that you tried to get a headstart. The bet is off!**"

4: We destroy the marker, the 'start point' of the race. It is removed from the game.

5-7: The cues 'countdown', 'coward' and 'go' and all of their sub-cues are cancelled. This effectively stops the mission from continuing.

In the next cue, a sub-cue of 'headstart', we're once more referencing the 'clean up' library cue, which will conveniently dispose of any game elements created for the mission.

```
<cues>
  <cue ref="clean up" comment="10 seconds from now the opponent will jump to
    another sector where it will be destroyed">
    <params>
1    <param name="time" value="10s"/>
    </params>
  </cue>
</cues>
</cue>      <- This is closing the 'headstart' cue.
```

We've seen this cue before, using the same time parameter [1]. We'll come back and look at this 'clean up' cue when we come to the library cue of the same name.

So, finally we get to see the intrepid pilots start this race.

```
<cue name="go">
  <condition>
1  <cue_is_complete cue="inposition"/>
  </condition>
  <timing>
2  <time min="7s"/>
  </timing>
  <action>
    <do_all>
3    <play_subtitles author="{object.pilot@createopponent.opponent}"
      text="{1081,10}"/>
4    <play_sound soundid="923"/>
5    <set_command object="createopponent.opponent" command="dock"
      commandobject="S01M25S00prepare.finish"/>
6    <destroy_object object="closetoplayer.marker"/>
    </do_all>
  </action>
```

- 1:** The actions will only trigger if the 'inposition' cue has been completed. It's pretty logical that they should be properly positioned before the race can begin properly.
- 2:** The actions of this cue will be triggered seven seconds after the conditions are met.
- 3:** The first of the actions is the playing of a subtitle on behalf of the opponent which simply displays 'GO!'
- 4:** Again we hear the 'alert' sound (sound ID 923) to signify the start signal.
- 5:** At the same time as the sound and subtitle occur, the opponent is given the command to dock at the station we earlier defined as being the 'finish' destination.
- 6:** Just by way of tidying up, the starting marker is removed from the game, since it's no longer needed.

The next cue is a sub-cue of 'go'.

```
<cues>
  <cue name="annoyme">
    <condition>
1     <cue_is_complete cue="go"/>
    </condition>
    <timing>
2     <count min="3" max="6"/>
3     <time min="30s" max="60s"/>
    </timing>
    <action>
      <do_all>
4       <set_value name="this.txtid" min="21" max="40"/>
5       <incoming_message popup="1" temporary="1"
          author="{object.pilot@createopponent.opponent}"
          text="{1081,{value@this.txtid}}"/>
      </do_all>
    </action>
6 </cue>
```

1: A simple condition to check whether the action(s) of the parent cue, 'go', has been completed.

2: The count here means that the actions will be carried out 3 to 6 times. The number is random.

3: At a random time between 30 and 60 seconds after the race starts the actions will be triggered.

4: Here we are setting the value of 'this.txtid' to between 21 and 40, again the actual number is random. So each time this line is read it will assign any number in that range to the value.

5: The net result of the <timing> node and the setting of the 'this.txtid' value is that we can send the player a pop-up message about 30-60 seconds after the start, selected randomly in each of three to six iterations from a range of messages (text IDs 21-40) held in the Text DB (Page 1081). The messages are marked as 'temporary', meaning that they will not be stored in the player log. Note the use in the text="" attribute of the value we assigned in **4**.

That's the end of that cue, but there is another sub-cue of 'go'.

```

<cue name="winner">
  <condition>
    <check_all>
1     <object_is_docked dockobject="S01M25S00prepare.finish"/>
2     <object_is_docked dockobject="S01M25S00prepare.finish"
      object="createopponent.opponent" negate="1"/>
    </check_all>
  </condition>
  <action>
    <do_all>
3     <cancel_cue cue="loser"/>
4     <cancel_cue cue="coward"/>
5     <cancel_cue cue="annoyme"/>
6     <play_sound soundid="1007"/>
7     <incoming_message author="{object.pilot@createopponent.opponent}"
      text="{1081,3}" popup="1"/>
8     <set_relation object="createopponent.opponent">
      <relation relation="friend" relationobject="{player.ship}"/>
    </set_relation>
9     <reward_player>
      <money exact="{value@S01M25S00prepare.reward}"/>
    </reward_player>
    </do_all>
  </action>

```

Let's deal with the conditions, **1** and **2** together. What it's saying here is that if the player is docked at the 'finish' station *and* the opponent is not docked there, then the actions will trigger. The negate="1" is checking if the condition is not true.

So, if those two conditions are met, there are few actions we need to happen.

3-5: The cues 'loser', 'coward' and 'annoyme' and their sub-cues are cancelled.

6: The sound '1007' is played (*(unused) DING DA DONG station announcement variation 2*)

7: The player receives a pop-up message from the opponent saying **'This is {object.pilot@createopponent.opponent} of the Argon. You win! Transferring {value@S01M25S00prepare.reward} Credits now.Let's have a drink at the bar!'**

8: Our next step is to reset the relation of the opponent to the player; as you will recall, we set him to 'enemy' status for the race. So now, a good loser, he's our friend again.....

9: ...so much so, he wants to give us our winnings. The value stated is referring back to the 'this.reward' declared in the **S01M25S00prepare** cue.

Of course that's what happens if the player wins. If the player fails to beat the opponent to the station then the next step is to look at the conditions in which the player loses... but first, it's 'clean up' time again.

```
<cues>
  <cue ref="clean up" comment="5 minutes from now the opponent will jump to another
sector where it will be destroyed">
  <params>
    <param name="time" value="5m"/>
  </params>
</cue>
</cues>
</cue>
```

<- The 'winner' cue is closed with this tag.

As in previous occurrences of the 'clean up' library cue this will neatly dispose of our mission components, although on this occasion the time parameter is set to a value of 5m (minutes). Now to the 'loser' cue:

```
<cue name="loser">
  <condition>
    <check_all>
1  <object_is_docked dockobject="S01M25S00prepare.finish" negate="1"/>
2  <object_is_docked dockobject="S01M25S00prepare.finish"
    object="createopponent.opponent"/>
    </check_all>
  </condition>
  <action>
    <do_all>
3  <cancel_cue cue="winner"/>
4  <cancel_cue cue="coward"/>
5  <cancel_cue cue="annoyme"/>
6  <play_sound soundid="1007"/>
7  <incoming_message author="{object.pilot@createopponent.opponent}"
    text="{1081,4}" popup="1"/>
8  <set_relation object="createopponent.opponent">
9  <relation relation="friend" relationobject="{player.ship}"/>
    </set_relation>
    <reward_player>
10 <money exact="-{value@S01M25S00prepare.reward}"/>
    </reward_player>
    </do_all>
  </action>
  <cues>
11 <cue ref="clean up" comment="5 minutes from now the opponent will jump to another
    sector where it will be destroyed">
    <params>
      <param name="time" value="5m"/>
    </params>
  </cue>
  </cues>
</cue>
```

<- This tag closes the 'loser' cue.

At first glance there doesn't appear to be much difference between the 'winner' cue and this one. Note also the 'loser' cue is also followed by the 'clean up' library cue [11], using the same time parameter as above.

1&2: Very similar to the conditions of the previous 'winner' cue, this time it's the check of the player having docked that is negated. At the same time the opponent's ship is

checked that it's docked. So, if the opponent gets there first the actions of this cue are triggered.

3-6: Almost exactly the same as the 'winner' cue, but the 'winner' cue is cancelled instead, along with the 'coward' and 'annoyme' cues (and their sub-cues). Also the sound triggered in 'winner' is played.

7: Again like 'winner' an incoming pop-up message is sent by the opponent to the player, but as you can imagine, the content is slightly different this time:

"This is {object.pilot@createopponent.opponent} of the Argon. That was close but I WIN. I'm the greatest, the Fastest and the Meanest! Where's the money!? Let me buy you a drink in the bar."

8&9: Like in 'winner', we now set the relation of the opponent back to friend.

10: In this <reward_player> node, we're doing the opposite of what we did in 'winner'; the player has lost the bet, so must pay up. That's why at the beginning we checked that the player has enough money to pay the bet and have 500Cr to spare.

Take a deep breath now and give yourself a pat on the back for making it this far. That's now the bulk of the mission finished. There's a little tidying up to be done and also we'll be taking a look at that library cue mentioned several times earlier. The crucial thing to remember about library cues is that they must be situated above or on the same level as the cues that refer to them. It's important to remember that 'above' here means in the cue structure rather than 'above' in the file structure. So although the library cues come right at the bottom of the file, they are above or at the same level as the <cue ref="blah"> that are referring to them. So to make sure we are placing the library cues in this file above those particular cues, we must close any cues that are in between.

<code></cue></code>	<- This tag is closing the 'go' cue.
<code></cues></code>	
<code></cue></code>	<- This tag is closing the 'inposition' cue.
<code></cues></code>	
<code></cue></code>	<- This tag is closing the 'closetoplayer' cue.
<code></cues></code>	
<code></cue></code>	<- This tag is closing the 'createopponent' cue.

So with those four cues closed we're now back up at the same level as the 'reset', 'selectOpponent' and 'createopponent' cues. This is where we want to put the library cue we've already referred to.

Now for the finale, as far as this mission is concerned... thankfully the last blocks of code for this mission. The library cue 'clean up' sits at the same level as the 'selectOpponent' and 'createopponent' cues and consists of a single cue with two sub-cues.

```
1<cue name="clean up" library="1">
  <timing>
2   <time min="{param@time}"/>
  </timing>
  <action>
    <do_all>
3     <find_sector name="this.jumpsec" exact="1"/>
4     <find_gate name="this.jumpgate" nearest="1">
        <sector sector="this.jumpsec"/>
      </find_gate>
5     <set_command object="createopponent.opponent" command="jumpsector"
        commandobject="this.jumpgate">
        <sector sector="this.jumpsec"/>
      </set_command>
    </do_all>
  </action>
```

- 1: To make a cue a library cue, and to ensure it's not parsed we have the library="1" attribute. We also give it the name 'clean up'; this, of course, is the name we used in the ref="" attribute in order to refer to the library cue and get it into action.
- 2: The time specified here {param@time} will use the time specified in the referencing cue.
- 3: Next we'll look for a sector exactly one jump away from our current one and call it 'this.jumpsec'.
- 4: Then we'll find a gate in the sector we've just found and call it 'this.jumpgate'.
- 5: Finally, in this cue, we give the opponent the command to jump to that sector using the gate we just found.

```
<cues>
  <cue name="destroy opponent">
    <condition>
1      <object_changed_sector object="createopponent.opponent" />
    </condition>
    <timing>
2      <time min="1s"/>
    </timing>
    <action>
3      <do_all>
        <destroy_object object="createopponent.opponent" />
      </do_all>
    </action>
  </cue>
```

- 1: If the opponent changes sector (which it should, as we've just told it to) the condition is met.
- 2: One second after the opponent has changed sector.....
- 3: ...he is destroyed in a big puff of pixels.

It's not over quite yet. The cuename and comment give it away a little

```
  <cue name="restart" comment="make the mission available again">
    <timing>
1      <time min="1h"/>
    </timing>
    <action>
2      <reset_cue cue="S01M25S00prepare"/>
    </action>
  </cue>
</cues>
</cue>    <- This is 'clean up' library cue being closed
</cues>
```

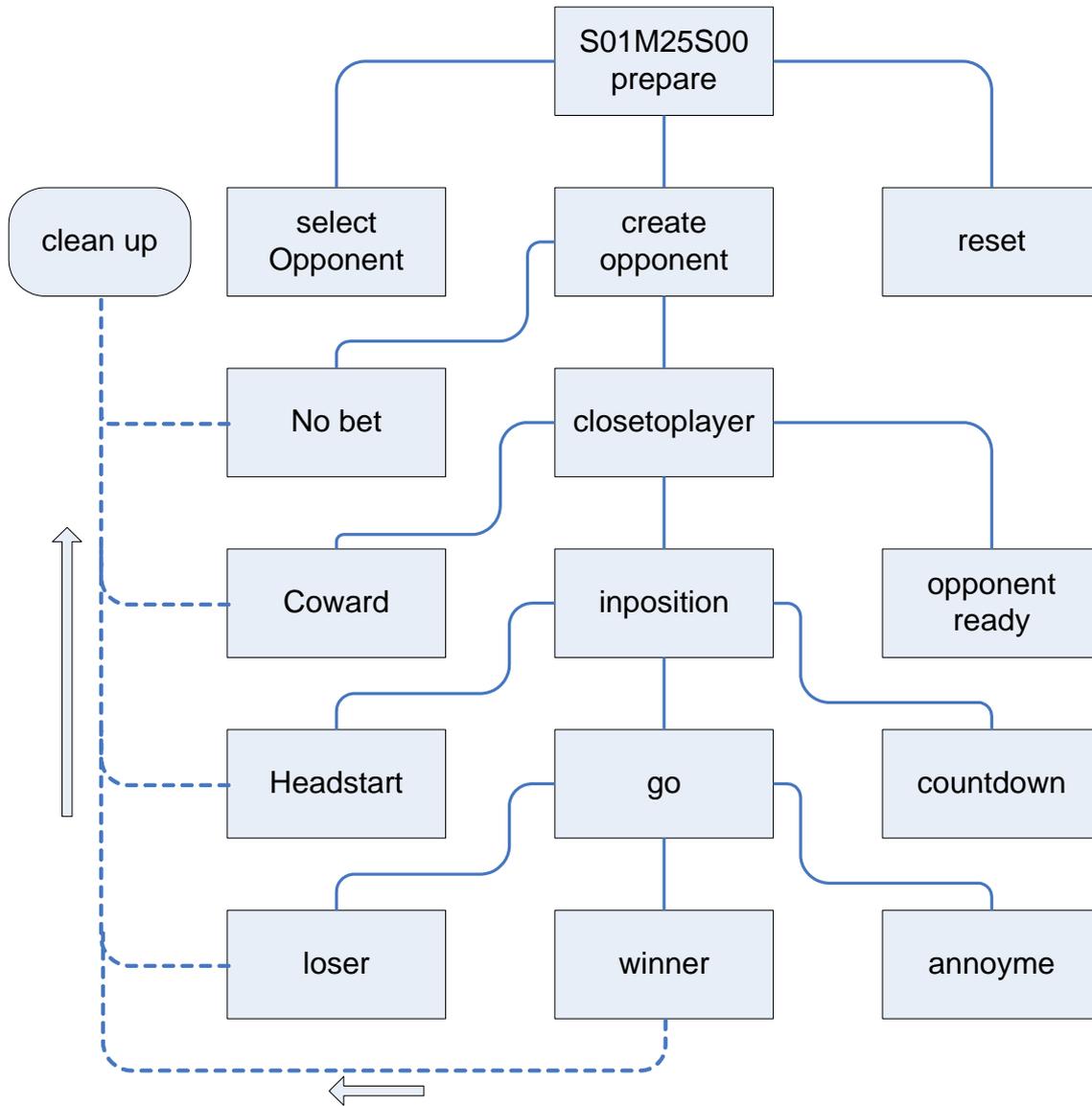
- 1: One hour after the opponent is destroyed (note that no conditions have to be met)...
- 2: the mission's top-level cue is reset and it all starts all over again.

Just the loose ends to tie up then... the very last tags in our mission file should be the opposite of the very first </cue>, </cues> and </director>.

```
</cue>    <- This is 'S01M25S00prepare' being closed
</cues>
</director>
```

So there we have it. One whole functioning mission.

If we were to visualise the structure of this mission, in a hierarchical tree structure it should look something like this:



As your mission progresses, keep a pencil and paper handy to draw a similar diagram of what your mission looks like. This will help you to remember relationships and where to insert new mission content, if required. Having a visualisation of what your mission is doing is very useful.

BBS MISSION-WRITING TUTORIAL

In the same way that we looked at an in-space mission, we should examine how to write a BBS mission, as there are some marked differences in the structure. A BBS template file is provided with the Mission Director documentation and is located in the *samples* folder, within the *director* folder. The essential elements of a BBS-based mission structure are contained in that file. It is worth studying this format and experimenting with the file before beginning your own mission.

First, we'll start with the header and documentation. This should have the same structure as any mission:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet href="director.xsl" type="text/xsl" ?>
<director name="BBSTemplate" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="director.xsd">
  <documentation>
    <author name="Fred Bloggs" alias="Froggs" contact="Froggs@egosoft.com"/>
    <content name="Lottery BBS" description="BBS-based lottery mission"
            reference="B01M47S00_XXXT"/>
    <version date="29 May 2007" number="1.0" status="Beta Test"/>
  </documentation>
</director>
```

As with our previous mission, it's essential that cue names are unique in order to avoid conflicts with other missions. It is suggested that cues are prefixed with a unique mission reference, for example: [B01M47S00MainCue](#)

```
<cues>
1 <cue name="B01M47S00LotteryMainCue" comment="This is the top-level cue, the start
    point for our quest">
  <condition>
2   <check_age value="{player.age}" min="5s" comment="Conditions to determine
    when, or where, or under what circumstances your BBS Mission should appear"/>
  </condition>
  <action>
3   <do_all>
4     <add_bbs_quest name="B01M47S00Lottery" priority="500" max="1"
        comment="Priority= How often your quest should occur on a scale of 0-100.
        Max= Max amount of the same quest per station"/>
     </do_all>
  </action>
</cues>
```

1: This is the introduction of our top-level cue, with a unique mission reference prefix and an appropriate comment. I have used the template, so you can see what each major element is doing.

2: The condition of this quest is currently a 'debug' one; this means that for testing purposes the action will be carried out after 5 seconds of player game time. Other conditions can of course be used to trigger a BBS mission.

3: Although there is only one action, a `<do_all>` has been used, just in case a further action is required at a later stage.

4: The action here is the most important one for the BBS mission. `<add_bbs_quest>`, as the name suggests, adds the quest to the pool of quests available to the BBS quest engine. The priority is the criterion, by which the quest engine will decide how often your quest will appear on BBS. The scale is technically 0-100 although using a priority of 500 will guarantee that the mission appears for testing purposes. *Max* determines the number of times your quest will be offered at the same station. Normally this is once. The next cue is a sub-cue of the top-level cue and here we will see what happens to our quest, once it is added.

```
<ques>
1 <cue name="B01M47S00Lottery Mission Offer" instantiate="static" comment="This will
   offer the Lottery Mission to the player">
  <condition>
2    <bbs_quest_evaluated quest="B01M47S00Lottery" comment="Will evaluate your quest
      by its given priority and max you set above"/>
  </condition>
  <action>
3    <offer_bbs_quest quest="B01M47S00Lottery" author="Spamelot Lottery"
      text="[center]Would you like to buy a lottery ticket and win more than a
        guaranteed 1,000,000Cr?\n\nThey cost only 1,000Cr!\n\n[select value='yes']You
        bet![/select]\n[select value='no']Forget it![/select][/center]"/>
  </action>
```

1: This cue is a sub-cue of the top-level cue. Remember that `<ques>` directly following `</action>` indicates a sub-cue. This is the cue, in which the BBS quest will be offered at stations. `instantiate="static"` means that a copy of this cue is created and the player will play the copy. Other copies will be available still at other stations. A detailed explanation of instantiation is available in *Appendix 1: Instantiation*.

2: Here, as a condition, we are telling the game to check if the quest has met the priority and *max* criteria we set in the parent cue. If those criteria are met then the action is triggered.

3: Next, the quest is offered in a station. Here we state the author and provide the text of the quest offer (of course in an official mission we should be using textIDs). So, for 1,000Cr we can win at least a million...

The next cue, again a sub-cue of the parent, determines what happens if the player says 'yes' to accept the quest.

```
<cue name="B01M47S00 Mission Accepted" comment="If the player accepts your quest">
  <condition>
    <check_all>
1      <bbs_quest_selected quest="B01M47S00Lottery" answer="yes"/>
    </check_all>
  </condition>
  <action>
    <do_all>
2      <do_choose>
3        <do_when value="{player.money}" min="1001">
4          <accept_bbs_quest quest="B01M47S00Lottery"/>
5          <reward_player>
              <money exact="-1000"/>
            </reward_player>
        </do_when>
6        <do_otherwise>
7          <incoming_message author="Spamelot Lottery" text="You cannot buy a ticket
              with money you don't have. Goodbye."/>
8          <cancel_cue cue="lucky dip"/>
        </do_otherwise>
      </do_choose>
    </do_all>
  </action>
```

1: If the quest was selected, i.e. the answer is yes, the action(s) will trigger

2: Here we see <do_choose> being used to good effect, it will have at least two sub-nodes, <do_when> and <do_otherwise>, the usage of the whole <do_choose> node can be likened to 'if x=y do a, else do b'.

3: So, 'if' the player has at least 1001Cr then...

4: This action then tells the quest engine that the quest is now active.

5: <reward_player> using a negative amount of money is the easiest way of charging the player for something.

6: Now the 'else' bit, if the player hasn't got at least 1001Cr then...

7: ...the player receives a message from the lottery company, chastising the player for trying to buy a ticket without the money.

8: To make sure that's the end of it if the player can't afford a ticket, the following cue and its sub-cues is cancelled.

```
<cues>
  <cue name="lucky dip">
    <condition>
1     <cue_is_complete cue="parent"/>
    </condition>
    <action>
      <do_all>
2       <set_value min="1" max="7" name="this.rand1"/>
3       <set_value min="8" max="17" name="this.rand2"/>
4       <set_value min="18" max="25" name="this.rand3"/>
5       <set_value min="26" max="33" name="this.rand4"/>
6       <set_value min="34" max="42" name="this.rand5"/>
7       <set_value min="43" max="49" name="this.rand6"/>
8       <incoming_message author="Spamelot Lottery" text="Your lucky dip numbers are:
          {value@this.rand1} {value@this.rand2} {value@this.rand3} {value@this.rand4}
          {value@this.rand5} {value@this.rand6}\n\nGood luck!"/>
      </do_all>
    </action>
```

1: The actions of this cue will be triggered once the parent cue's actions are completed.

2-7: These lines set the values of the named variables to a random number within the provided ranges.

8: An incoming message, using the values set above, is sent to the player telling them what their lottery numbers are. How the numbers are generated isn't vitally important, since the odds of winning are calculated by other means.

This cue is a further sub-cue of 'lucky dip'; here we see the draw itself.

```

<cue>
  <cue name="draw" comment="a certain amount of time after ticket purchase">
    <timing>
1     <time min="20s" max="25s"/>
2     <!--The 20-25s is just a debug time to shorten testing -->
3     <!--<time min="10h" max="12h"/>-->
    </timing>
    <action>
      <do_all>
4       <set_value name="this.prize" min="1000000" max="1500000" comment="Prize is
          somewhere between 1 and 1.5 million Cr"/>
5       <set_value name="this.lotterynum" min="1" max="100000" comment="the odds of
          winning are 1 in 100000"/>
      <do_choose>
6       <do_when value="{value@this.lotterynum}" max="1">
7         <incoming_message author="Spamelot Lottery" text="Congratulations! You've
          won the jackpot!\n\n{value@this.prize} Credits are being transferred
          to your account."/>
8         <reward_player>
          <money exact="{value@this.prize}"/>
        </reward_player>
      </do_when>
9       <do_otherwise>
10      <incoming_message author="Spamelot Lottery" text="Sorry, you have not won
          a prize this time. Please try again soon."/>
      </do_otherwise>
    </do_choose>
  </do_all>
</action>
</cue>
</cues>

```

1-3: Looking at the timing, we've got one range of timings for debug/testing purposes. In the final version, it would be between 10 and 12 hours after the ticket is bought that the draw takes place.

4: Here we are assigning to the variable {this.prize} a random value between 1 and 1.5 million credits.

5: Here we're setting the winning odds. We've assigned a random value of 1 to 100,000 to the name 'this.lotterynum'. 1 in 100,000 is therefore the chance of winning.

6: In another <do_choose>, we're saying 'if' the value of 'this.lotterynum' is 1 (i.e. if that number in 100,000 comes up) then the enclosed actions will be carried out.

7: If the 1 chance in 100,000 comes up, the player receives a congratulatory message and notification of the transfer of winnings. {value@this.prize} the random number set above is transferred into the player's account.... Nice.

8: It's so good to be able to add money to the player account and this time we're adding a positive number to the <reward_player> node, in this case the same value that we used in the message above.

9&10: So, if any number between 2 and 100,000 is evaluated (ie the 'otherwise' is the case, the player will receive a consolation message.

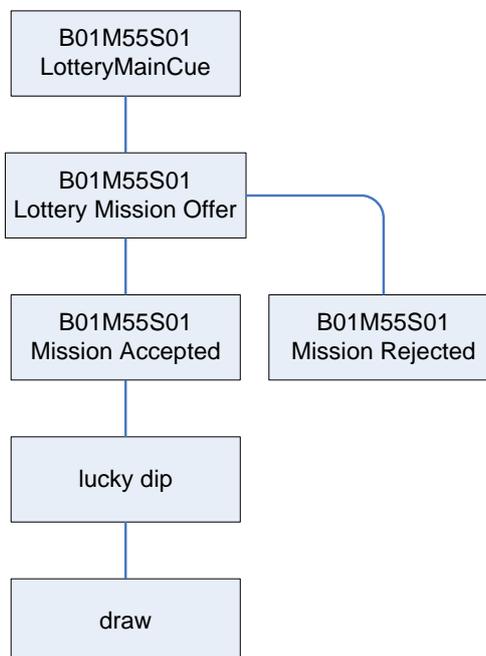
```

        </cue>          <- This is 'lucky dip' being closed.
    </cues>
    </cue>          <- This is 'B01M47S00 Mission Accepted' being closed.
1   <cue name="B01M47S00 Mission Rejected" comment="If the player declines
    your quest">
2     <condition>
        <bbs_quest_selected quest="B01M47S00Lottery" answer="no" />
    </condition>
    <action>
3     <do_all>
        <incoming_message author="Spamelot Lottery Services" text="Too bad!
            You look the lucky type." />
    </do_all>
    </action>
    </cue>
    </cues>
    </cue>          <- This is 'B01M47S00Lottery Mission Offer' being closed.
    </cues>
    </cue>          <- This is 'B01M47S00LotteryMainCue' being closed.
    </cues>
</director>

```

- 1: If the player's original response to the offer is no, this cue deals with that eventuality. Note that it's on the same level of the cue structure as B01M47S00 Mission Accepted.
- 2: As with the mission accepted cue, if the cue's condition is evaluated as 'no' then an action is performed.
- 3: The action is simply an incoming message telling the player what they've missed out on.

That then is a BBS mission. A fairly simple one admittedly, but it contains all of the key features that are required. Let's take a look at the cue structure and you can see how that compares with the structure that exists in the .xml file.



A VARIABLE WORTH KNOWING ABOUT...

The variable, {group.object.select@group} is a useful one to know. If your mission requires the player to select one of a number of grouped objects in a message, then this is what you should use. The first thing you need to do is define the objects in the group.

To illustrate how this works let's find a group of stations in the player's current sector:

```
<action>
  <do_all>
    1 <find_station class="station" race="{player.sector.race.name}" max="5"
      group="Foundstations" multiple="1">
      <sector sector="{player.sector}" />
    </find_station>
  </do_all>
</action>
```

1: In the find_station sub-node we are looking for a maximum of 5 stations, owned by the sector's owner race, located (as per the sector sub-node) in the player's sector. We are assigning the group name 'foundstations' to the result of the search. 'Multiple' specifies that the result of the search must be more than one object.

Now in a subsequent message, in which one of the objects needs to be selected, we can use {group.object.select@group}.

```
<action>
  <do_all>
    1 <do_if min='1' value='{group.object.count@Foundstations}'>
    2 <offer_bbs_quest quest="Bargains"
      author="{random.pilot@{player.sector.race.name}}" text="Here is a list of
      stations, at which you can buy the specified ware. Select one now.\n\n
      [center]{group.object.select@Foundstations}[/center]" />
    </do_if>
  </do_all>
</action>
```

1: Here we're checking that the number of stations found as a result of our search is at least one.

2: If so, the BBS quest, 'Bargains', is offered. The author is a random name of the race of the sector the player is currently in. We are telling the player in the message that there are up to 5 stations at which the player can buy a certain ware.

```
<cue name="BBS Mission Accepted">
  <condition>
    1 <bbs_quest_selected quest="Bargains" />
  </condition>
```

1: Normally the <bbs_quest_selected> sub-node requires both quest and an answer attribute. Here there is no answer attribute; instead, the object selected is the answer and the quest continues with that object in mind.

To be able to make further use of that selected object in the BBS quest we should create the selection as a game object.

```
<action>
1 <set_object name="this.selection" value="{quest.answer@Bargains}"/>
</action>
```

1: Here we assign the name 'this.selection' to the object selected from the BBS Offer. Note the quest's name in the variable following the @ sign.

That object 'this.selection' can now be the object or target of whichever commands are required for the mission to progress.

This functionality extends to list selection not in BBS missions. The creation of the group, from which an object will be selected, remains the same. Once an object has been selected from the list, it must be made into a usable game object. That's done in the same way as above, but with a slight difference.

```
<action>
1 <set_object name="this.selection" value="{question.answer@missiles}"/>
</action>
```

1: Note here the only difference is that it's the answer of the question rather than the quest, which we want to make into an object.

This means that you must use the <ask_question> sub-node and that you must specify the question's name in the value above. For example:

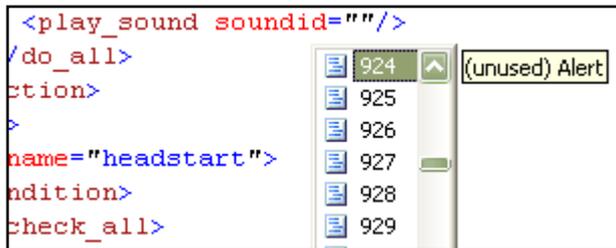
```
<action>
  <do_all>
    <find_station class="mine" max="10" group="mines" multiple="1">
      <sector sector="{player.sector}"/>
    </find_station>
    <ask_question name="select" author="Security" text="There are up to 10 mines in
      this sector requiring destruction. Please select the first mine you intend
      to destroy so we can clear the area of routine traffic.
      n\n[center] {group.object.select@mines} [/center]"/>
  </do_all>
</action>
< cues>
  <cue name="identification">
    <condition>
      <question_answered question="select"/>
    </condition>
    <action>
      <do_all>
        <set_object name="this.selection" value="{question.answer@select}"/>
        <set_target object="this.selection"/>
      </do_all>
    </action>
  </cue>
```

INCORPORATING IN-GAME SOUNDS

As you saw in the mission writing tutorial any number of various in-game sounds can be played under a variety of circumstances. One of the examples you will have come across is:

```
<play_sound soundid="924"/>
```

Incorporating in-game sound like that is very easy. When you select the 'soundid' attribute from the lookup list, a long list of numbers appears. The description of the sound relative to each number is displayed in the lookup:



The list of sound IDs and their descriptions can also be found in the *director.htm* documentation file.

INCORPORATING TEXTS

Many examples of incorporating texts have been used in the tutorial missions. In incoming messages, BBS offers, questions you can include free-flow text for your missions. Developers are urged to keep their texts as brief as possible. Also, as suggested in the tutorial, official missions will use textID references in the code, rather than free-flow.

```
So, instead of using texts like this in our official missions...
<incoming_message author="Shipboard Computer" text="Hello World"/>

We should be using text references like this...
<incoming message author="{1323,119}" text="{1278,305}"/>
```

TESTING YOUR MISSIONS

It is imperative that you test your own mission as its development progresses. This is a key way to ensure correct functionality at each stage. By testing it regularly and see how different conditions and actions affect the mission, you can make changes, sometimes in real time, to correct or change the mission's behaviour.

The Mission Director Menu



The Mission Director menu is accessed by pressing the Enter key on the keypad for the main menu and selecting Mission Director, as illustrated in the screenshot.

This menu allows you to do two things. Firstly it allows you to view your mission cues. Secondly, and most importantly, it allows you to reset the Mission Director in-game. Please note though, that this does not work with BBS quests and

objects created in previous test iterations will still be present in the game. If you need a clean slate with which to test, it's best to start a new custom game.

As mentioned earlier, all of the .xml mission files located in the *director* folder will be loaded into the game. It is wise, when testing a new mission that no other mission files are in the folder to interfere with its operation. The 'director' files, of course, must remain in that folder.

As with the basic test earlier, it is best to start a new game in a custom universe. Once loaded the game should respond to the initial starting condition of the mission. If it appears that that is not the case, then you should quit the game (to the main menu), make appropriate changes then start a new game in a custom universe.

Using the *director.dmp* file

This file, located in the game's root directory, can be an invaluable source of information when troubleshooting and debugging a mission.

The dump file is composed of three sections.

- A rather chaotic-looking first section is the file being read and the XML being parsed.
- The second section is a neater rendering of the cues and their constituent elements.
- The final section gives the user a time-indexed account of game events and notifications, resulting from the conditions and actions of each cue as it is completed.

The first thing you should look for when encountering difficulties is any occurrence of the word 'error'. Seeing where that occurs will help you to diagnose the problem. Noting the last entries in the file will often show you where the mission is halting, even if there is no specific code problem stopping it from working.

When testing your mission you'll know what cue should be running at any given stage, so by checking that particular cue's entry in the dump file, you can see if a condition is not being met or an action not being performed. If that cue does not appear at all, then it is likely that the problem lies with the parent cue. If the cue appears (usually as "Conditions checked (first time)") you know the cue is there and the conditions are actually checked - see if it appears again so you know whether the conditions are being met. If it doesn't appear after that you can be pretty certain that's because the conditions have not been met - you've narrowed the problem down to the condition of the cue in question.

If you're working with values (global or local) the dump will tell you what value has been assigned to it (and also if it's been assigned) that helps to see if maybe some calculations went wrong. If you are not certain about the values you are using and how they might appear, you can add, for debug purposes only, an incoming message to output the values you are working with. Also very useful for event debugging is that all MD-relevant events that occur are listed in the dump, so if you are checking for an event and the condition doesn't trigger or is met at an unexpected time, check the dump to see when (or if) the event occurred.

Isolation Testing

If you have multiple conditions and you are unsure at which point the mission is failing, it makes sense to perform an 'isolation test'. That is, to reduce them one by one to find the one that is causing problems. That means putting that cue and its condition in a new file (make sure the cue names are different from the original) and using that to test if the conditions can be matched (if necessary create a parent cue to set some things you need to the appropriate values).

GLOSSARY

Action – a key node in a cue; this defines what happens in the game once certain conditions have been met.

Attribute – provides additional information about the node or sub-node with which it is associated.

Condition – a key node in a cue; this sets conditions to be met or events that have to have occurred for certain actions to take place.

Cue – a key element of Mission Director mission structure, containing condition, timing and action information.

Expression – a mathematical operation using numbers and/or variables.

Instance – a copy of a cue, created by the instantiate attribute, separate to the original.

Internal ID – each object in the game has an internal ID that uniquely identifies that object.

Library cue – a cue containing one or more other cues of conditions and actions, which can be called upon as needed, using ref="". They are not parsed otherwise.

Look-up – a dropdown menu of possible values for an attribute.

Node – an XML tag that performs a specific function.

Object – in the Mission Director, this is a game object, already existing or created by the mission.

Parsing – this means essentially 'reading and processing', i.e. how the game interprets the Mission Director XML.

Quest – a mission, whether BBS, HQ, In-space or Plot-related.

Schema – an XML file that provides the structure for nodes, sub-nodes and look-ups.

Sub-cue – a cue which exists only inside another cue for which the conditions of the parent cue are implicit, i.e. it will not even be created until those conditions are met.

Sub-node – a sub-node is similar in function to an attribute but groups several additional pieces of information.

Timing – a key node in a cue, this establishes elapsed time between conditions being met and actions taking place, the number of times the actions will be performed, and the intervals between them being performed.

Value – this can be any piece of data; a string, an integer, a number, a variable. The game uses many values in many ways.

Variable – a value (text or number), which can be used or created to substitute variable values into missions rather than only being able to use fixed values.

APPENDICES

In the appendices you'll find some useful information which is a bit long and heavy to be squeezed in the main body of the guide. Enjoy.

APPENDIX 1: INSTANTIATION

Instantiation changes what happens when a cue's conditions are met - if a cue is NOT instantiated then the cue's actions are run (however many times are specified in the timing) and the cue is marked as completed - if a cue IS instantiated then a COPY of the cue (and all its sub-cues) is made and it is this COPY in which the actions are performed and it is the COPY whose status is set to complete when they are finished - this means that if the conditions in an instantiated cue are met again then the whole thing happens all over again.

An instantiated cue should only be used with conditions that are only going to be met once (or a fairly limited number of times) OR with conditions that include an event condition.

Instantiation should NOT be used in a cue which, say, just depends on the game time being greater than a specific value as this will result in a copy of the cue being made every few milliseconds and the game crashing shortly afterwards.

The most common use of an instantiated cue is in responding to events such as the player ship changing sector or a BBS mission being evaluated.

Because the copying process copies a cue AND all its sub-cues, you need to be "instance-aware" in any cue which has an ancestor that is instantiated. instance-aware means being careful about the scope of stored values, objects, cue names, etc., being aware that the original cue will always appear not to have been activated, and being aware that your instance will not just be marked as complete when it has finished but will also actually cease to exist when all of its sub-cues and their sub-sub-cues etc have also been marked as complete.

It is also possible to have an instantiated sub-cue within an instantiated sub-cue - i.e. you can end up with one or more copies of parts of a cue structure that is itself a copy of the original - this sounds complicated but means that you can, for example, have an instantiated cue that responds to player sector changes in each of several active copies of the same mission (which are instances because they are coming from evaluating BBS offer conditions).

APPENDIX 2: LIBRARY CUES

It is possible to create reusable library cues. For example, if you currently have several cues that create a random M5-class ship, you no longer need to copy and paste the code and remember to change each copy whenever you want to improve it. Instead you can create a single cue, marked as a library cue so that it does not get triggered in the normal way, and refer to it from each of the cases where you want to use it.

For example:

```
<cue name="mylibrarycue" library="1">
  ...
</cue>

<cue name="myfirstcue">
  ...
  <cues>
    <cue ref="mylibrarycue" />
  </cues>
</cue>

<cue name="mysecondcue">
  ...
  <cues>
    <cue ref="mylibrarycue" />
  </cues>
</cue>
```

In the above example, 'mylibrarycue' itself would be ignored because it is marked as a library, but its definition would replace the two references in the sub-cues of 'myfirstcue' and 'mysecondcue', and it would be triggered normally in both of the two cues in which it is referred to.

Sometimes it is useful to use the same complex cue conditions to trigger different actions or have different cue conditions trigger the same action. In these cases a library cue can still be used, by referring to the cue from only the component(s) you want replacing with the library version. For example:

```
<cue name="mylibrarycue" library="1">
  <action>
    ...
  </action>
</cue>

<cue name="myfirstcue">
  <condition>
    ...
  </condition>
  <action ref="mylibrarycue" />
</cue>

<cue name="mysecondcue">
  <condition>
    ...
  </condition>
  <action ref="mylibrarycue" />
</cue>
```

Finally, library cues are all very useful but sometimes you want cues that are almost but not quite the same. For example, you might want a cue that creates a random M5 class ship and sets it to be owned by the target race in your mission. You could create it in a library cue and then set ownership separately, but that could get quite messy. A simpler solution is to use a parameter in your library cue, as follows:

```
<cue name="mylibrarycue" library="1">
  <action>
    ...
    <create_ship ...="" race="{param@targetrace}" ...="" >
    ...
  </create_ship>
  ...
</action>
</cue>

<cue name="myfirstcue">
  ...
  <cues>
    <cue ref="mylibrarycue">
      <params>
        <param name="targetrace" value="{value@myfirstcue.myrace}" />
      </params>
    </cue>
  </cues>
</cue>

<cue name="mysecondcue">
  <condition>
    ...
  </condition>
  <action ref="mylibrarycue">
    <params>
      <param name="targetrace" value="{lookup.race@boron}" />
    </params>
  </action>
</cue>
```

Notice how in this example, one cue is using the library cue as a complete sub-cue and the other is using just the action component of it.

Because no parameter type is specified it will accept the 'targetrace' parameter from either source. If you want to be more specific about where a library cue should and should not take its parameters from then you can specify this in the parameter variable, e.g. "{param.cue@targetrace}" or "{param.action@targetrace}", in which case the parameter will only evaluate if the library was referred to in the corresponding way. You can pass as many parameters as you like to a library cue, and you can pass ordinary values (like "1" or "m5") as well as variables.

You can define a library cue at the top level of the cue structure in a director file or inside another cue - in the latter case the scope of a library cue is that it is visible to all cues from the cue in which it is defined downwards in the cue tree. It is not valid for a top-level cue to be a reference, but components of a top-level cue can be references provided the library cue they refer to is global in scope (i.e. also top-level). Finally, a cue does not absolutely have to have a library flag in order to be used as a library. You can "borrow" a normal cue or cue component for use in another cue, regardless of that flag, provided it is within scope.