A First Draft of an XML Declaration for Patterns

Darrell Ferguson and Dwight Deugo
School of Computer Science
Carleton University
Ottawa, Ontario, K1S 5B6
(darrell.ferguson@ottawa.com, deugo@scs.carleton.ca)

Abstract: In this manuscript, we propose an XML declaration for the specification of patterns using XML.

Introduction

As the pattern community continues to grow, the documentation of new or proto patterns is becoming a problem. The representation of a pattern changes from author to author, making it difficult for readers and reviewers to understand the patterns. There is a need for an adaptable structure to represent patterns, which is easy for the pattern's author and its potential readers to use. The use of XML (Extensible Markup Language) is a step towards solving the problem. XML is a markup language for documents that contain structured information. With patterns being represented using structured information, XML is the logical choice for structuring them. By considering XML, the problem now shifts to 'what is the structure of a pattern?' Not only do we need to come up with a universal structure of patterns, it must also be versatile enough to incorporate the needs of future patterns formats.

This manuscript proposes a first draft of an XML declaration for patterns. Whether this declaration will meet the needs of pattern authors and readers is left for debate. However, with the growth of XML and patterns, the declaration's flexibility and ability to adapt to the needs of future authors is a good start.

Pattern Components

In determining the XML declaration for a pattern, it becomes necessary to assess the components of a pattern. This leads us to question when is a pattern a pattern? What does a pattern need to be a pattern? What is a required part of a pattern and what is not? What components do we need individualized and what components may or may not be combined? These are all difficult questions to answer. The XML declaration must be able to adapt to individual style and preferences and yet be structured in such a way that others may understand the pattern quickly and effectively.

We propose that pattern components are separated into two categories, mandatory and optional, similar to the ones proposed by Meszaros and Doble [Meszaros+98]. There may be varying names for each component (and this report will attempt to clarify what the sections are and what they may also be known as) but the underlying meanings are the same.

Mandatory Components

Title

All patterns must have a title for referencing. One could just imagine attempting to communicate about patterns without being able to identify the pattern in question. Some may refer to this as the name of the pattern.

Authors

Every pattern has been written so it only seems logical that there was a writer or a group of writers and credit should be given to the people who do the work. Some pattern books assume the author or editor of the book is the author but we are not given this luxury in our XML documents.

Context

The context contains the overall circumstances in which the problem has occurred and imposes constraints on the solution. It could be described by an explicit statement or by a situation. The context may also contain in it, individual forces to be considered when designing the solution. This accommodation was made to allow authors to write more readable while still highlighting the forces involved. These individual forces should be considered an extension of the forces section of the pattern.

Problem

All patterns solve a problem in a context. If the problem did not exist then the notion of designing a pattern becomes ridiculous. The statement of the problem should be separate from the constraints on the system (the context). The problem description may also contain in it, individual forces to be considered when designing the solution. These individual forces should be considered an extension of the forces section of the pattern.

Forces

The forces section shows what considerations are to be accounted for when deciding on a solution to the problem. This section should also show more clearly the ideas behind the development of the pattern. An accommodation which must be made in regards to the forces section is the desire to highlight the forces within the pattern description (either the problem or context section).

Solution

This section contains the proposed solution to the problem. It is critical for this section to be clear to the reader, as it is the climax of the pattern. Although some pattern writers will divide this section up into other sections such as structure, participants, collaborators, implementation, sample code, etc., for the moment, we will combine it into one component.

Optional Components

Symptoms

This section will explain any preexisting symptoms that may be seen and signal that the stated problem exists. The constant need to recompile a system to add a subclass of an object may be a symptoms which indicates that an example of the Type Object pattern [Martin+98] should be used.

Resulting Context

As the name implies, this section describes the state after the application of the pattern. One or more patterns can be included as solutions to new problems that arise.

Related Patterns

Included in this list are other possible solutions to the problem, variations of the pattern and patterns that relate to the resulting context.

Known Uses

The application of the pattern can sometimes be a little confusing if not illustrated. The known uses section gives us a convenient location to place concrete examples that illustrate the use of the pattern.

Sample Code

Actual code (usually in C++ or Smalltalk) showing an implementation of the pattern.

Rationale

This section covers the justification of why the presented solution is most appropriate solution relating back to the stated problem and the context of the problem.

Aliases

These will be other names that the presented pattern might be known.

Acknowledgments

Just as one should acknowledge the author of a pattern, anyone who contributed to the development of a pattern should be given recognition as well. Exactly who deserves to be in the acknowledgments will be left to the author or editor of the pattern.

References

The references section will list any relevant material that was used in writing the pattern. These can published patterns, books, websites, etc.

XML Declaration

This section describes the XML declaration for patterns. We will discuss the issues encountered, the possible solutions and the chosen solutions. We will also examine possible extensions that can be made. It will be vital for anyone using the XML declaration for documenting their patterns to read and understand this section. One note: if a pattern writer wishes that a section appears as a different heading other than the given name, they can set the label attribute of the tag (as seen in the second example of title).

Pattern

The pattern must be contained within the tags <pattern> and </pattern>. A pattern must consist of the required components listed above in the required components section. They must occur in one of these orders: title, authors, problem, context, forces, and solution or title, authors, context, problem, forces and solution. For examples of these or possible changes to this structure through the use of optional sections, please see the following Pattern Components section.

The positioning of optional components is restricted in terms of required components but not in terms of each other. This means that it is valid to have a solutions section followed by a resulting context section which is followed by related patterns or, you may reverse the order and place related patterns in front of the resulting context. Although in this case, it would seem wiser to place resulting context first, this decision is left to the writer of the pattern.

Pattern Components

Title

The title of the pattern must be the first element of the pattern. The title element can only contain text (no images, bullets, or any other component). Two examples of titles are:

```
<title>Null Object</title>
<title label=”Name”>Visitor</title>
```

Authors

The author must be the second component of the pattern. The authors element can contain only author elements (and must contain at least one author) which may contain text along with emails and websites (no images, bullets, or any other component). A full description of the email and url tags can be found in the general components section of this document. Three examples of authors are:

```
1)
    <authors><author>Dwight Deugo</author></authors>

2)
    <authors label=”Editor”><author>Gang of Four</author></authors>
```

```
3)
    <authors><author>Darrell Ferguson
    <email>darrell.ferguson@ottawa.com</email>
    <url>http://www.nexus.carleton.ca/~dferguso</url></author>
    <author>Dwight Deugo
    <email>dwight@scs.carleton.ca</email>
    <email>deugo@scs.carleton.ca</email>
    <url>http://www.scs.carleton.ca/~deugo</url>
    </author></authors>
```

Context

The context section may occur before or after the problem section. The context may contain all general components listed in the general components section of this document.

Problem

The problem section may occur before or after the context section. This section may contain all general components listed in the general components section. The problem section may also contain individual forces. Here is an example of a problem section with forces embedded in it:

```
<problem>How do you build an XML Declaration for Patterns which <force>is
extensible</force> and <force> is easy to use</force></problem>
```

Forces

The forces section (not to be mistaken for the force sections contained in the problem section) must occur after at least one of the problem or context sections. Despite the introduction of the force components inside the problem section, the forces section is required (if the forces are listed in the problem section than the user may enter “Refer to Problem Section”). A possible improvement to this would be to give the sections an attribute visibility attribute which can be set to ‘visible’ or ‘hidden’ to allow future viewers to leave out the forces section.

The forces section may contain all general components listed in the general components section of this document. Some examples of a forces section are:

```
<forces>Although UML is known by some designers, it is not understood by everyone.
Documentation must be made clear to everyone</forces>
```

or:

```
<forces><bullets><bullet>Computer Scientists use different languages</bullet>
<bullet>Some languages are more suitable to certain problems</bullet></bullets>
</forces>
```

Solution

The solution section must come after the forces section. The solution section may contain all components listed in the general components section of this report. A possible addition to the solution section would be to add common subsections.

Symptoms

The symptoms section is an optional section which may appear some place between the problem, context or forces section or after all three, before the solution section. It may contain all general components listed in the general components section of this document.

Resulting Context

The resulting context section is an optional section that may appear after the solution and before any ending sections (i.e. references or acknowledgments). It may contain any combination of elements from the components listed in the general components section of this document. The tag for resulting context is <resultingContext>.

Related Patterns

The related patterns section is an optional section which may appear after the solution and before any ending sections (i.e. references or acknowledgments). It may contain any combination of elements from the components listed in the general components section of this document. The tag for related patterns is <relatedPatterns>.

Known Uses

The knownUses section is an optional section that may appear after the problem, context and forces sections and before the solution section or between the solution section and any ending sections (i.e. references or acknowledgments). It may contain any combination of components listed in the general components section of this document. The known uses tag is <knownUses>.

Sample Code

The sample code section is an optional section that may appear between the solution section of the pattern and any ending sections (i.e. references or acknowledgments). It may contain any combination of components listed in the general components section of this document. The tag for the sample code section is <sampleCode>.

This section is not to be mistaken for the code section section. Where the code section is code and pseudocode, the sample code section code samples as well as explanations and/or diagrams illustrating the design.

Rationale

The rationale section is an optional section that may appear between the solution section of the pattern and any ending sections (i.e. references or acknowledgments). Any combination of the general components can be contained in the rationale section. The tag for the rationale section is <rationale>.

Aliases

The aliases section is an optional section that may appear between the authors section and the problem, context, and forces sections or, after the problem, context and forces sections and before the solution section. The aliases section may contain any combination of the general components listed in this document. The tag for the aliases section is <aliases>.

Acknowledgments

The acknowledgments section is an optional section that may appear at the end of the pattern (after solution and all solution-related sections). The acknowledgments section may contain any combination of the general components listed in this document. The tag for the acknowledgments section is <acknowledgments>.

References

The references section is an optional section that may appear at the end of the pattern (after solution and all solution-related sections). The references section may contain any combination of the general components listed in this document. The tag for the references section is <references>

Miscellaneous Section

The miscellaneous section is an optional section that may appear where any of the above optional sections may appear. It can appear multiple times throughout the document. It is a section that the writer has felt necessary to put into the documentation of the pattern. It should never be assumed that a pattern contains a miscellaneous section and these sections can be ignored by applications such as pattern searchers and pattern verifiers. The label attribute of any miscellaneous section is required and should be used as the title when displaying or processing the section. An example of a miscellaneous section is:

```
<miscSection label=”Explanation”>Here is an explanation of the pattern</miscSection>
```

General Components

Text

Text may appear in many of the above components and does not need a special tag. A paragraph tag, <p/>, may be used to separate paragraphs in the text. A line break tag,
, may be used to indicate a new line. Inserting any other tags is possible as long as the outlying markup allows it, and this will automatically separate the text. The paragraph tag is not necessary at the end of a pattern section and should be left out. An example of text inside the problem section is:

```
<problem>How do you build a DTD for patterns?<p/>Some considerations must be made for
several areas.<figure label="Fig 1.1.1" src="fig1-1-1.jpg">Consideration
diagram</figure>However, only one DTD can be decided upon.</problem>
```

There are actually three sections of text in the above example, these would be the underlined portions. The paragraph and line break tags may be documented internally (for the benefit of the pattern author only) by using the label and role attributes. The values assigned will not be used by any outside application nor be apparent to the reader. An example of this would be:

```
<p label="Solution paragraph" role="Explain the general structure"/> The general
structure of the components …
```

Text may also be stylized through the use of italic and bold tags (<italic> and <bold> respectively). Text, bullet groups, individual bullets, and entire miscellaneous subsections may be placed within the style tags as well as the style tags themselves (in order to make the text bold and italicized). As an example, we have:

```
The <bold>Bold</bold> and the <italic>Beautiful</italic> is a <bold><italic>soap
opera</italic></bold>
```

When writing the document, there should be no formatting of the text using actual returns and tabbing. Returns will be ignored by applications processing the document (including the viewer) and tabs could cause the document to appear poorly when using the viewer. Any line breaks the author wishes to enforce should be done using the line break tag or the paragraph tag and tabbing for bullets and other elements should be done by the viewing application.

Figure

The figure tag should be used for generic figures/diagrams. The label attribute is required for any figure and should resemble ‘Figure 1.1’ or some other meaningful name. The src attribute is required as well and should be the name of the file where the image is stored. The text between the figure tags will be the description for the figure. An example of the figure markup is:

```
<figure label="Figure 1.1" src="./fig1-1.jpg">Collection class and subclasses</figure>
```

Table

The table tag should be used for tables that are presented in the pattern. The label attribute is required for any table and should resemble ‘Table 1.1’ or some other meaningful name. The src attribute is required as well and should be the name of the file where the image of the table can be found. The text between the table tags will be the description used when displaying the table. An example of a table is:

```
<table label="Table 1.1" src="./table1-1.gif">Efficiency of Smalltalk compared to
C++</table>
```

Graph

The graph tag should be used for graphs that are presented in the pattern. The label attribute is required for any graph and should resemble ‘Graph 1.1’ or some other meaningful name. The src attribute

is required as well and should be the name of the file where the image of the graph can be found. The text between the graph tags will be the description used when displaying the graph. An example of a graph is:

```
<graph label="Graph 1.1" src="./graph1-1.gif">The breakdown of message type sent to
the proxy object in pie chart form</graph>
```

Bullets

Bullets can be used to create a list of generic content. The bullets may be given a label but any description of the list should be placed before the bullets. Bullets must be composed of individual 'bullet's. An example of using bullets (in the forces section) is:

```
<forces>Some considerations must be given for:
<bullets><bullet>Efficiency of the system is vital</bullet>
<bullet>The class structure becomes too complex if unwatched.  An example of this is:
<figure label="Figure 1.2" src="fig1-2.jpg">The class structure for the
pattern</figure> </bullet></bullets></forces>
```

For future designs, we may wish to consider introducing specialized bullets including adding an index. This should be done by not changing bullets but adding a different type of bullet (i.e. indexedBullet) and allowing bullets to contain these different types.

Bullet

A bullet can only occur within bullets and may contain any generic component listed in this section of the document. For an example of bullet, please see bullets. A bullet may be given a label as well. This label will be automatically formatted by the viewing application. This is done with the label attribute and appears as:

```
<bullet label="Initial documentation.">  Is a requirement…</bullet>
```

Force

A force section (not to be confused with the forces section in a pattern) may contain any generic content. The force sections may only occur inside the problem or context section of a pattern. The tag for a force section is <force>.

Code Section

The code section tag (not to be confused with the sample code section in a pattern) contains code and/or pseudocode. No regular text or images or any other general component should appear in a code section. Currently, only internal code sections are available although external code samples may be allowed at a later time. The tag for internal code sections is <internal.codeSection> and the body must begin with '<![CDATA[' and end with ']]>' to avoid parsing the code section (which may lead to XML errors). An example of an internal code section is:

```
<internal.codeSection><![CDATA[if ((a+b) < 6) return c;]]></internal.codeSection>
```

Author

The author tag must be used in the authors section of the pattern. It may consist of text (the author's name), emails (one or more) and urls (one or more). For examples of the author tag, please refer to the Authors component in the Pattern Components section of this document.

Email

The email tag is optional section that can be placed only in the authors section of the document. It consists of the email name only between the start and end tags and any web-based viewer should convert this to the appropriate html tags. An example of an email tag is:

```
<email>darrell.ferguson@ottawa.com</email>
```

Url

The url tag is also an optional section that can be placed only in the authors section of the document. It consists of the url only between the start and end tags and any web-based viewer should convert this to the appropriate html tags. An example of an url tag is:

```
<url>http://www.nexus.carleton.ca/~dferguso</url>
```

Miscellaneous Subsection

The miscellaneous subsection is a section that may appear inside of one of the components listed in the Pattern Component section of this document. The miscellaneous subsection will help pattern writers break up their sections but the breaks should be ignored by some applications (i.e. a pattern verifier). The label of a miscellaneous subsection is required. A miscellaneous subsection can contain any combination of general components listed in this section of this document. An example of a miscellaneous subsection is:

```
<miscSubsection label=”Collaborators”>The Collaborators sections.</miscSubsection>
```

General XML Documentation

Header Information

The following text must be placed at the beginning of the XML document for processing instructions (assuming that the file “pattern.dtd” is in the same directory as the XML document, if it is not, then the full path must be stated).

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE pattern SYSTEM "pattern.dtd">
```

Each component must begin with its start tag (unless otherwise stated, this is the component name, in lower case, enclosed between ‘<’ and ‘>’) and end with its end tag (the same as the start tag except it start with ‘</’ and ends with ‘>’).

Using the DTD

Finding an XML parser is required in order to use the pattern declaration. There are many XML parsers in existence and can be found on the web. A validating parser is highly recommended if you wish to verify that the documented pattern is valid and not just that it is well formed.

In order to confirm that the documented pattern is valid, the document type declaration (DTD file) is required. It can be stored locally or accessed via a URL but it must be made clear in the header where in the header of the XML document.

Tools

The pattern declaration is very limited if there were no tools that took advantage of its potential. Some tools currently being worked on are: an editor to create, edit and validate pattern documents, a viewer to view pattern documents in a viewer friendly format (a standalone version and a web application), and a pattern repository for storing and viewing documented patterns and searching. Further documentation of these tools will be made available as soon as possible (http://muffin.nexus.carleton.ca/~darrell/repo/).

Conclusion

The XML declaration for patterns described in this manuscript gives structure to the rapidly expanding world of patterns while leaving room for individuality and adaptive behavior. We have used our declaration to structure a number of patterns written using different formats, such as GOF and POSA. As an example of the adaptability and potential of the structure, we have written the Singleton pattern using our XML declaration. We believe that XML and the associated pattern DTD will allow patterns to be verified and processed more efficiently and effectively. This will aid not only the writers of patterns but

also, anyone looking to use patterns. In addition, the specification will make the previously stated tools possible.

In giving a structure to patterns, the writers of patterns may perceive a restriction on their pattern design. Any structure that was put to the declaration was seen as required and with the use of optional sections, miscellaneous sections, and re-labeling of components, the freedom still exists for writers while providing a structure to patterns that can be used by various applications.

As with everything, the evolution of patterns continues to grow, and it is hoped that this pattern declaration will evolve with future requirements. The possibility for expansion and change exists, along with the freedom for the writer and the structure required for more effective processing.

References

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA.: Addison-Wesley, 1995.

[Martin+98] R. Martin, D. Riehle, F. Buschmann. *Pattern Languages of Program Design 3*. Reading, MA.: Addison-Wesley, 1998.

[Meszaros+98] G. Meszaros, and Jim Doble, *A Pattern Language for Pattern Writing*, In Pattern Languages of Program Design 3, eds. R. Martin, D. Riehle, F. Buschmann , Reading, MA.: Addison-Wesley, 529-574, 1998.

Appendix A

Singleton Pattern Example

Below is an example of the XML declaration in practice. It is the Singleton pattern from [Gamma+95]. It should be pointed out that a change was made to the Singleton Pattern. The Consequences section was moved from its position in Design Patterns book (immediately following the Collaborations section) to just after the Sample Code section. This is to make use of the Resulting Context section of the XML Declaration.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE pattern SYSTEM "pattern.dtd">
<pattern>
<title>Singleton</title>
<authors label="Editors"><author>Erich Gamma</author>
<author>Richard Helm</author>
<author>Ralph Johnson</author>
<author>John Vlissides</author>
</authors>
<miscSection label="Category">Object Creational</miscSection>
<problem label="Intent">Ensure a class only has one instance, and provide a global point
of access to it</problem>
<forces label="Motivation">It's important for some classes to have exactly one instance.
Although there can be many printers in a system, there should be only one printer
spooler.  There should be only one file system and one window manager.  A digital filter
will have one A/D converter.  An accounting system will be dedicated to serving one
company.<p/>
How do we ensure that a class has only one instance and that the instance is easily
accessible?  A global variable makes an object accessible, but it doesn't keep you from
instantiating multiple objects.<p/>
A better solution is to make the class itself responsible for keeping track of its sole
instance.  The class can ensure that no other instance can be created (by intercepting
requests to create new objects), and it can provide a way to
access the instance.  This is the Singleton pattern.
</forces>
<context label="Applicability">Use the Singleton pattern when
<bullets>
```

```
	<bullet>there must be exactly one instance of a class, and it must be accessible
to clients from a well-known access point.</bullet>
	<bullet>when the sole instance should be extensible by subclassing, and clients
should be able to use an extended instance without modifying their code.</bullet>
</bullets>
</context>
<solution>
<miscSubsection label="Structure">
<figure label="Singleton Structure" src="single1.gif">The class structure for the
Singleton pattern</figure>
</miscSubsection>
<miscSubsection label="Participants">
<bullets><bullet><bold>Singleton</bold>
<bullets><bullet>defines an Instance operation that lets clients access its unique
instance.  Instance is a class operation (that is, a class method in Smalltalk and a
static member function in C++).</bullet>
<bullet>may be responsible for creating its own unique instance.</bullet>
</bullets>
</bullet>
</bullets>
</miscSubsection>
<miscSubsection label="Collaborations">
<bullets>
<bullet>Clients access a Singleton instance solely through Singleton's Instance
operation.</bullet>
</bullets>
</miscSubsection>
<miscSubsection label="Implementation">Here are implementation issues to consider when
using the Singleton pattern:
<bullets>
<bullet label="Ensuring a unique instance.">  The Singleton pattern makes the sole
instance a normal instance of a class, but that class is written so that only one
instance can ever be created.  A common way to do this is to hide the operation that
creates the instance behind a class operation (that is either a static member function or
a class method) that guarantees only one instance is created.  This operation has access
to the variable that holds the unique instance and it ensures the variable is initialized
with the unique instance before returning its value.  This approach ensures that a
singleton is created and initialized before its use.<p/>
You can define the class operation in C++ with a static member function
<italic>Instance</italic> of the <italic>Singleton</italic> class.  Singleton also
defines a static member variable <italic>_instance</italic> that contains a pointer to
its unique instance.<p/>
The <italic>Singleton</italic> class is declared as
<internal.codeSection>
<![CDATA[
	class Singleton {
	public:
		static Singleton* Instance();
	protected:
		Singleton();
	private:
		static Singleton* _instance;
	};
]]>
</internal.codeSection>
<internal.codeSection>
The corresponding implementation is
<![CDATA[
	Singleton* Singleton::_instance = 0;

	Singleton* Singleton::Instance() {
		if (_instance == 0) {
			_instance = new Singleton;
		}
		return _instance;
	}
]]>
</internal.codeSection>
Clients access the singleton exclusively through the <italic>Instance</italic> member
function.  The variable <italic>_instance</italic> is initialized to 0, and the static
member function <italic>Instance</italic> returns its value, initializing it with the
```

unique instance if it is 0. <italic>Instance</italic> uses lazy initialization; the value it returns isn't created and stored until it's first accessed.<p/>
Notice that the constructor is protected. A client that tries to instantiate <italic>Singleton</italic> directly will get an error at compile-time. This ensures that only one instance can ever get created.<p/>
Moreover, since the <italic>_instance</italic> is a pointer to a Singleton object, the Instance member function can assign a pointer to a subclass of Singleton to this variable. We'll give an example of this in the SampleCode.<p/>
There's another thing to note about the C++ implementation. It isn't enough to define the singleton as a global or static object and then rely on automatic initialization. These are three reasons for this:
<bullets>
<bullet>We can't guarantee that only one instance of a static object will ever be declared.</bullet>
<bullet>We might not have enough information to instantiate every singleton at static initialization time. A singleton might require values that are computed later in the program's execution.</bullet>
<bullet>C++ doesn't define the order in which constructors for global objects are called across translation units [ES90]. This means that no dependencies can exist between singletons; if any do, then errors are inevitable.</bullet>
</bullets>
An added (albeit small) liability of the global/static object approach is that it forces all singletons to be created whether they are used or not. Using a static member function avoids all these problems.<p/>
In Smalltalk, the function that returns the unique instance is implemented as a class method on the Singleton class. To ensure that only one instance is created, override the new operation. The resulting Singleton class might have the following two class methods, where <italic>SoleInstance</italic> is a class variable that is not used anywhere else:
<internal.codeSection>
<![CDATA[
	new
		self error: 'cannot create new object'

	default
		SoleInstance isNil ifTrue:[SoleInstance := super new].
		^ SoleInstance
]]>
</internal.codeSection>
</bullet>
<bullet label="Subclassing the Singleton class."> The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it. In essence, the variable that refers to the singleton instance must get initialized with an instance of the subclass. The simplest technique is to determine which singleton you want to use in the Singleton's <italic>Instance</italic> operation. An example in the Sample Code shows how to implement this technique with environment variables.<p/>
Another way to choose the subclass of Singleton is to take the implementation of Instance out of the parent class(e.g., <italic>MazeFactory</italic>) and put it in the subclass. That lets a C++ programmer decide the class of singleton at linktime (e.g., by linking in an object file containing a different implementation) but keeps it hidden from the clients of the singleton.<p/>
The link approach fixes the choice of singleton class at link-time, which makes it hard to choose the singleton class at run-time. Using conditional statements to determine the subclass is more flexible, but it hard-wires the set of possible Singleton classes. Neither approach is flexible enough in all cases.<p/>
A more flexible approach uses a <bold>registry of singletons</bold>. Instead of having Instance define the set of possible Singleton classes, the Singleton classes can register their singleton instance by name in a well-known registry.<p/>
The registry maps between string names and singletons. When <italic>Instance</italic>needs a singleton, it consults the registry, asking for the singleton by name. The registry looks up the corresponding singleton (if it exists) and returns it. This approach frees <italic>Instance</italic> from knowing all possible Singleton classes or instances. All it requires is a common interface for all Singleton classes that includes operations for the registry:
<internal.codeSection>
<![CDATA[
	class Singleton {
	public:
		static void Register(const char* name, Singleton*);
		static Singleton* Instance();
	protected:
		static Singleton* Lookup(const char* name);

```
        private:
                static Singleton* _instance;
                static List<NameSingletonPair>* _registry;
        }
]]>
</internal.codeSection>
Register registers the Singleton instance under the given name.  To keep the registry
simple, we'll have it store a list of <italic>NameSingletonPair </italic>objects.  Each
<italic>NameSingletonPair</italic> maps a name to a singleton.  The
<italic>Lookup</italic> operation finds a singleton given its name.  We'll assume that an
environment variable specifies the name of the singleton desired.
<internal.codeSection>
<![CDATA[
        Singleton* Singleton::Instance () {
                if (_instance == 0) {
                        const char* singletonName = getenv("SINGLETON");
                        // user or environment supplies this at startup

                        _instance = Lookup(singletonName);
                        // Lookup returns 0 if there's no such singleton
                }
                return _instance;
        }
]]>
</internal.codeSection>
Where do Singleton classes register themselves?  One possibility is in their constructor.
For example, a <italic>MySingleton</italic> subclass could do the following:
<internal.codeSection>
<![CDATA[
        MySingleton::MySingleton() {
                // ...
                Singleton::Register("MySingleton", this);
        }
]]>
</internal.codeSection>
Of course, the constructor won't get called unless someone instantiates the class, which
echoes the problem the Singleton pattern is trying to solve!  We can get around this
problem in C++ by defining a static instance of <italic>MySingleton</italic>.  For
example, we can define
<internal.codeSection>
<![CDATA[
        static MySingleton theSingleton;
]]>
</internal.codeSection>
in the file that contains <italic>MySingleton's</italic> implementation.<p/>
No longer is the Singleton class responsible for creating the singleton.  Instead, its
primary responsibility is to make the singleton object of choice accessible in the
system.  The static object approach still has a potential drawback - namely that
instances of all possible Singleton subclasses must be created, or else they won't be
registered.
</bullet>
</bullets>
</miscSubsection>
</solution>
<resultingContext label="Consequences">The Singleton pattern has several benefits:
<bullets>
<bullet label="Controlled access to sole instance.">  Because the Singleton class
encapsulates its sole instance, it can have strict control over how and when clients
access it.</bullet>
<bullet label="Reduced name space.">  The Singleton class may be subclassed, and it's
easy to configure an application with an instance of this extended class.  You can
configure the application with an instance of the class you need at run-time.</bullet>
<bullet label="Permits a variable number of instances">The pattern makes it easy to
change your mind and allow more than one instance of the Singleton class.  Moreover, you
can use the same approach to control the number of instances that the application uses.
Only the operation that grants access to the Singleton instance needs to change.</bullet>
<bullet label="More flexible than class operations.">  Another way to package a
singleton's functionality is to use class operations (that is, static member functions in
C++ or class methods in Smalltalk).  But both of these language techniques make it hard
to change a design to allow more than one instance of a class.  Moreover, static member
```

functions in C++ are never virtual, so subclasses can't override them polymorphically.</bullet>
</bullets>
</resultingContext>
<sampleCode>
Suppose we define a <italic>MazeFactory</italic> class for building mazes as described on page 92. <italic>MazeFactory</italic> defines an interface for building different parts of a maze. Subclasses can redefine the operations to return instances of specialized product classes, like <italic>BombedWall </italic>objects instead of plain <italic>Wall</italic> objects.<p/>
What's relevant here is that the Maze application needs only one instance of a maze factory, and that instance should be available to code that builds any part of the maze. This is where the Singleton pattern comes in. By making the <italic>MazeFactory</italic> a singleton, we make the maze object globally accessible without resorting to global variables.<p/>
For simplicity, let's assume we'll never subclass <italic>MazeFactory</italic>. (We'll consider the alternative in a moment.) We make it a Singleton class in C++ by adding a static <italic>Instance</italic> operation and a static <italic>_instance</italic> member to hold the one and only instance. We must also protect the constructor to prevent accidental instantiation, which might lead to more than one instance.
<internal.codeSection>
<![CDATA[
```
	class MazeFactory {
	public:
		static MazeFactory* Instance();

		// existing interface goes here
	protected:
		MazeFactory();
	Private:
		Static MazeFactory* _instance;
	};
```
]]>
</internal.codeSection>
The corresponding implementation is
<internal.codeSection>
<![CDATA[
```
	MazeFactory* MazeFactory::_instance = 0;

	MazeFactory* MazeFactory::Instance () {
		if (_instance == 0) {
			_instance = new MazeFactory;
		}
		return _instance;
	}
```
]]>
</internal.codeSection>
Now let's consider what happens when there are subclasses of<italic> MazeFactory</italic>, and the application must decide which one to use. We'll select the kind of maze through an environment variable and add code that instantiates the proper <italic>MazeFactory</italic> subclass based on the environment variable's value. The <italic>Instance</italic> operation is a good place to put this code, because it already instantiates <italic> MazeFactory</italic>:
<internal.codeSection>
<![CDATA[
```
	MazeFactory* MazeFactory::Instance() {
		if (_instance == 0) {
			const char* mazeStyle = getenv("MAZESTYLE");

			if (strcmp(mazeStyle, "bombed") == 0) {
				_instance = new BombedMazeFactory;
			} else if (strcmp(mazeStyle, "enchanted") == 0) {
				_instance = new EnchantedMazeFactory;

			// ... other possible subclasses

			} else {		// default
				_instance = new MazeFactory;
			}
		}
		return _instance;
```

```
       }
]]>
</internal.codeSection>
Note that <italic>Instance</italic> must be modified whenever you define a new subclass
of <italic>MazeFactory</italic>.  That might not be a problem in this application, but it
might be for abstract factories defined in a framework.<p/>
A possible solution would be to use the registry approach described in the Implementation
section.  Dynamic linking could be useful here as well - it would keep the application
from having to load all the subclasses that are not used.
</sampleCode>
<knownUses>
An example of the Singleton pattern in Smalltalk-80 [Par90] is the set of changes to the
code, which is <italic>ChangeSet current</italic>.  A more subtle example is the
relationship between classes and their <bold>metaclasses</bold>.  A metaclass is the
class of a class, and each metaclass has one instance.  Metaclasses do not have names
(except indirectly through their sole instance), but they keep track of their sole
instance and will not normally create another.<p/>
The InterViews user interface toolkit [LCI+92] uses the Singleton pattern to access the
unique instance of its Session and WidgetKit classes, among others.  Session defines the
application's main event dispatch loop, stores the user's database of stylistic
preferences, and manages connections to one or more physical displays.  WidgetKit is an
AbstractFactory (87) for defining the look and feel of user interface widgets.  The
<italic>Widget::instance()</italic> operation determines the particular WidgetKit
subclass that's instantiated based on an environment variable that Session defines.  A
similar operation on Session determines whether monochrome or color displays are
supported and configures the singleton Session instance accordingly.
</knownUses>
<relatedPatterns>
Many patterns can be implemented using the Singleton pattern.  See Abstract Factory (87),
Builder (97), and Prototype (117).
</relatedPatterns>
</pattern>
```