Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5707--00-8473

A Generic Preference System Pattern and C++ Implementation

GREGORY STERN
MICHAEL PILONE
BRIAN SOLAN

ENEWS Program
Tactical Electronic Warfare Division

September 29, 2000

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 29, 2000	Final

4. TITLE AND SUBTITLE

A Generic Preference System Pattern and C++ Implementation

5. FUNDING NUMBERS

PE – 0602270N
PR – EW70103

6. AUTHOR(S)

Gregory Stern, Michael Pilone, and Brian Solan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5707--00-8473

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Most applications require that users can customize features. This report presents a preference design pattern that solves that requirement. The design pattern provides two main features: generic GUI editing and serialization. The report also defines possible C++ implementation issues and proposes possible solutions. The Appendix contains a partial code listing for a C++ implementation of the design pattern.

14. SUBJECT TERMS

Preferences, Templated, Serialization, C++, RTTI, Generic, Conversion

15. NUMBER OF PAGES

79

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

Contents

A GENERIC PREFERENCE SYSTEM

Section 1: Introduction

Most applications require user configurable settings. For example, users often want to specify values such as default save directory path, the user's name, default window sizes, application flags, etc. The goal of this report is to define a design pattern along with a C++ implementation to satisfy the recurring need for a preference system. Before explaining the preference pattern and C++ implementation, a more thorough understanding of the application user's and software engineer's requirements is needed. These requirements will help to define exactly what is meant by the phrase "a preference system."

Users Requirements

Application users are usually not concerned about implementation of a preference system but rather the functionality that it provides them. Users require that they can easily edit individual preferences, or *properties*. They also want a friendly user interface that reasonably groups and describes the properties. These changes are generally expected to be persistent. In many cases, an application user would also like to change preferences for an application without having to actually use the application. Changing preferences independent of the application is helpful for reasons such as speed, support, and solving problems with the preferences that maybe having adverse effects on the application.

Engineers Requirements

Software engineers are concerned about effects of the preference system on the overall design of the application, difficulty and speed of implementation, readability, and ease of debugging. Software engineers want a system that will minimize repetitious redesigning and rewriting of common code across applications. Although they do not want to write repetitious code, software engineers still want flexibility for each application domain. Specifically an individual application may need some customizations in displaying the GUI. The

Manuscript approved July 12, 2000.

preference file format also may change among each application. The software engineer must also be guaranteed type safety when loading information or accepting information from a user to maintain application stability

Report Overview

The preference system we created satisfies both the application users' functionality requirements and the software engineers' implementation and impact requirement.

This report is split up into two main sections: the Preference Pattern and C++ Implementation of the Preference pattern[1].

Section 2: Preference pattern will describe the preference system design pattern, pattern applicability, use of the preference system, its structure, participant descriptions, participant collaboration, and consequences of the preference system.

Section 3: C++ Implementation of Preference Pattern details C++ implementation issues and possible solutions for the preference system.

The Appendix section contains a partial code listing for the Special Projects Group's implementation of this preference system, including a generic editor widget for the preference files. It also contains contact information to request the entire library.

Formatting Conventions

Throughout this report, the following format or style conventions are used.

- Class and function names are signified by a special font such as Class Name.
- Class diagrams follow the Unified Modeling Language format and syntax.
- Referenced section headings will be shown in *italics*.

[1] The format of the pattern and implementation sections use the same general outline as patterns presented in Design Patterns: Elements of Object-Oriented Software.

Section 2: Preference Pattern

The subsections covered are *Applicability*, *Using the Preference Pattern*, *Structure*, *Participant Collaborations*, *Consequences*, and *Pattern Remarks*.

- The *Applicability* subsection describes and summarizes when to use the preference pattern.
- The *Using the Preference Pattern* subsection provides an overview of how the users of the completed library would use the library to meet an individual application's needs.
- The *Structure* subsection provides UML diagrams describing the classes, or participants, in implementing the pattern.
- The *Participant Descriptions* subsection compliments the diagrams with a description of the overall purpose of the main classes.
- The *Participant Collaborations* subsection describe how the classes in the pattern work together.
- The *Consequences* subsection describes possible benefits and liabilities of the design pattern.
- The *Pattern Remarks* subsection describes how the preference pattern is actually composed of several basic patterns.

Applicability

Use the preference pattern to:

- Generically be able to create, edit and save a set of properties. Properties are simple types of information such as string, integer, and date.
- Guarantee type safety in editing and loading of properties.
- Save into user defined formats such as plain text, XML or binary.
- Allow for different conversion formats i.e. simple text, SQL.
- Organize properties into groups and subgroups.
- Create a graphical user interface (GUI) to edit and browse categorized preferences.

Using the Preference Pattern

Once the preference pattern has been implemented, using the preference library is a relatively simple task. The pattern can be used to store and retrieve preferences, create a GUI editor, and serialize the preferences[2].

Storing and Retrieving Preferences

All storage and retrieval functionality is provided in a class called Prefs. The Prefs class has two main methods for storage and retrieval: set()and get().

The set() method stores a preference into the class by associating the preference value with a variable name. The preference values can also be categorized by passing a group name into the set() method as well. A C++ code example of how to set values may look like:

```
// create the preference object
  Prefs prefs;
// store 1000 into a variable "maxWidth" in group Windows
prefs.set("maxWidth", 1000, type_id(int), "Windows", "Maximum width of a
                window");
// store "" into a variable "userName" in group Personal
prefs.set("userName", std::string(""), type_id(std::string), "Personal", "User
                name");
```

In this simple example, a Prefs object is created and two variables are stored in the object. The parameters passed to the set() method are variable name, value, type information, group name, and description. Don't worry about type information for now since it is a C++ implementation issue.

The get() method can be used to retrieve the preference. The get() method takes the name of the variable, the destination variable, type information and the group as parameters. A C++ code example that builds upon the example above may look like:

```
int width;
QString userName;
// retrieve the value associated with "maxWidth" into width
prefs.get("maxWidth", width, type_id(int), "Windows");
prefs.get("userName", userName, type_id(QString), "Personal");
```

[2] Serialization is the process of saving and loading of an object.

GUI Editor and Serialization

Creating a graphical preference editor and serializing the Prefs class can be accomplished with a few method calls. The preference editor in the example below is a dialog that takes the preference class. The code to show the editor could be as simple as:

```
Prefs prefs;
...
// preference items have been initialized
PrefsEditorDialog dialog(prefs, parentWindow);
dialog.show();
```

The dialog will edit the preferences generically. This dialog could be used inside an application or could in fact become its own application.

The Prefs class serialization is almost as simple. There is a class called PrefsSerializerRegistry that allows the specification of multiple file formats. The PrefsSerializerRegistry given the type name of the format returns a PrefsSerializer. Loading preferences is as simple as:

```
Prefs prefs;
fstream inFile("prefs.xml");
PrefsSerializer &ps =
  PrefsSerializer::instance().getPrefsSerializer( "XML" );
try {
ps.load( prefs, inFile );
}
catch (Exception e)
{ ... }
```

Saving is almost exactly the same except the call to PrefsSerializer would be save() instead of load().

Structure

An overview of the design that implements the preference system is shown in the UML diagram in Figure 1.

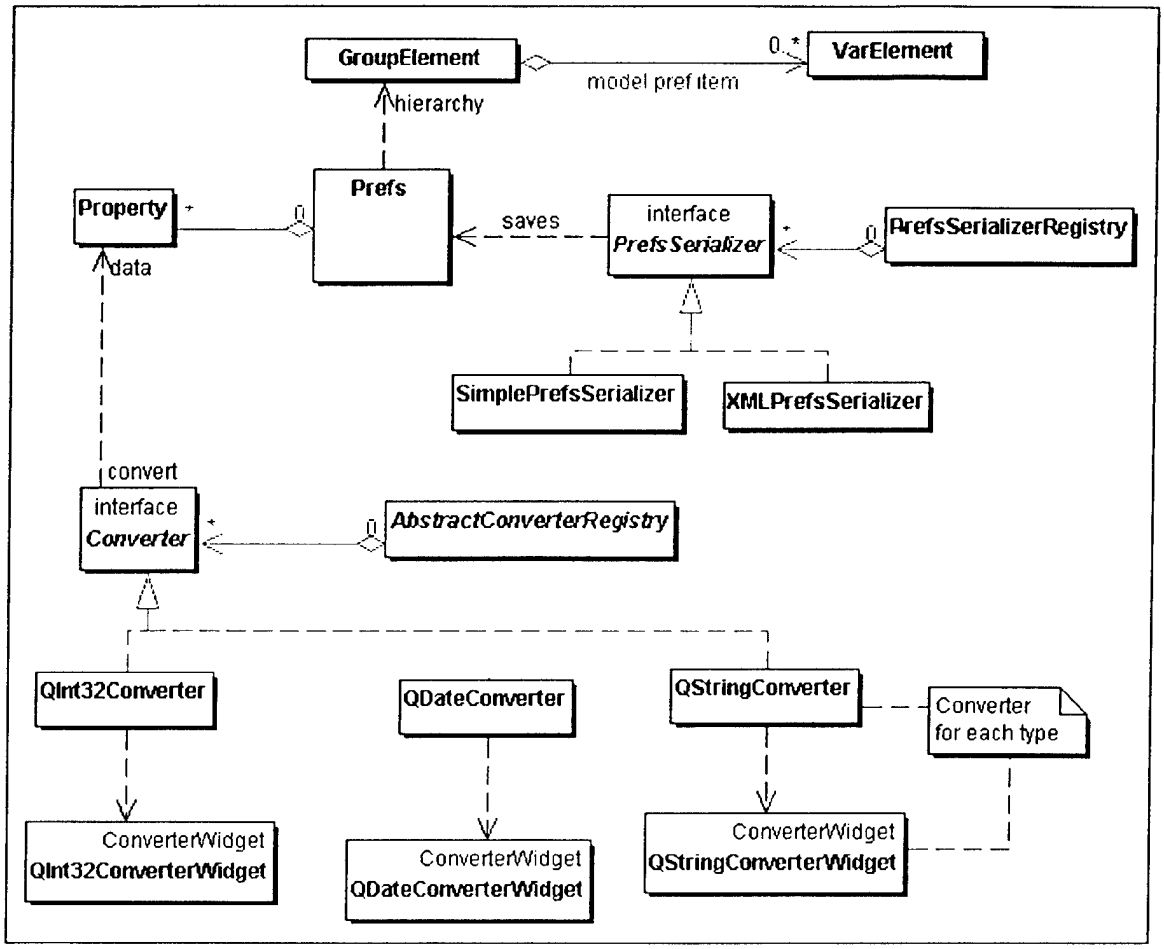

Figure 1: Preference Pattern Structure Overview

The preference system contains a core set of classes that control the storage and conversion of preferences as shown in Figure 2 below.

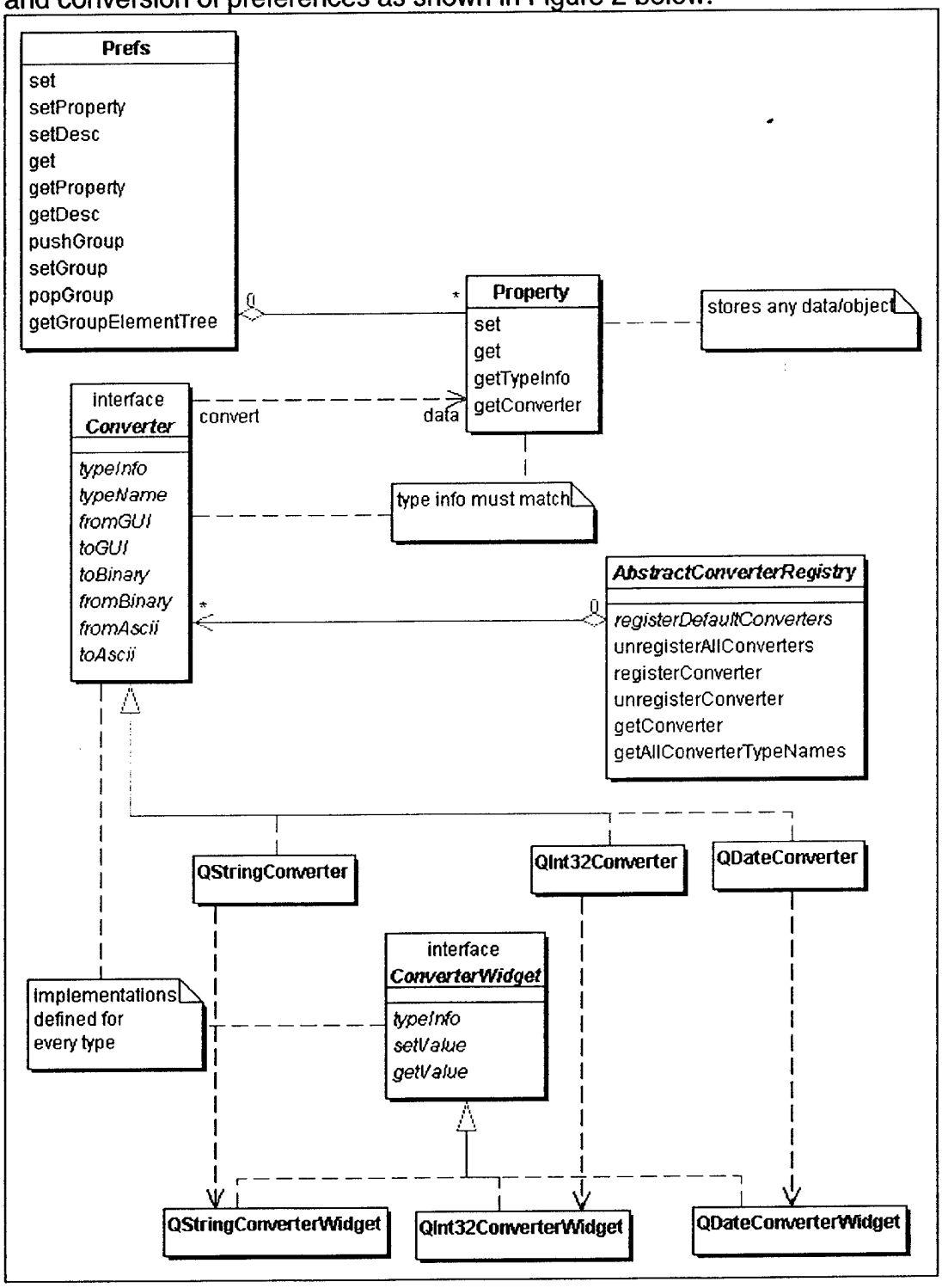

Figure 2: Preference Pattern Basic Structure

The Prefs class can generate a group/variable tree from the stored preferences. Subclasses of the PrefsSerializer interface can use the group/variable tree to serialize the Prefs class into any user defined format as shown in Figure 3 below.

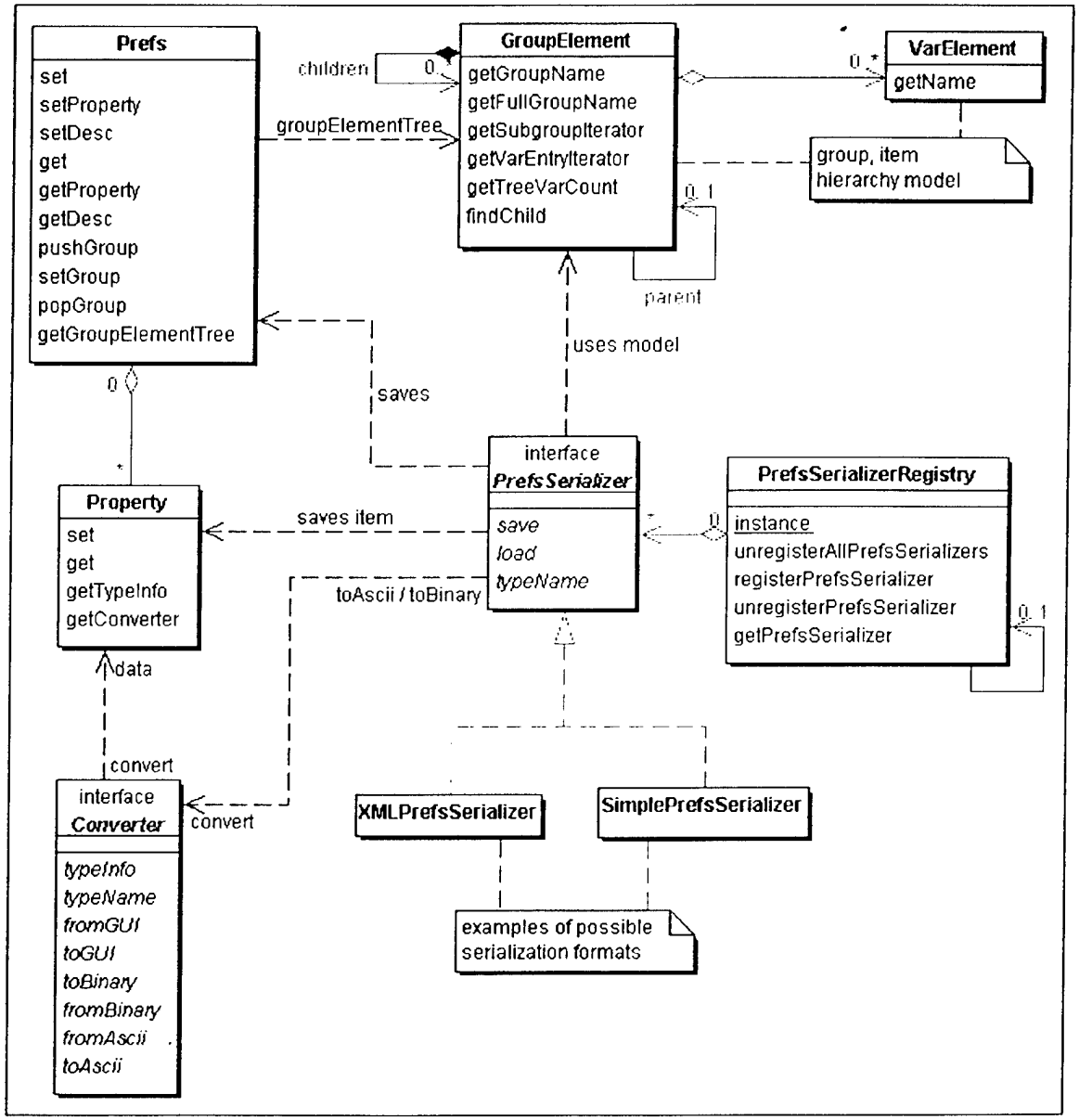

Figure 3: Preference Pattern Serialization

The PrefsEditor allows the user of the application to graphically edit the Prefs class. The PrefsEditor is displayed in a PrefsDialog as shown in Figure 4: Preference Editor Pattern. The editor also uses the group/variable tree to build and layout the editor components. The Converters provide the individual editor components.

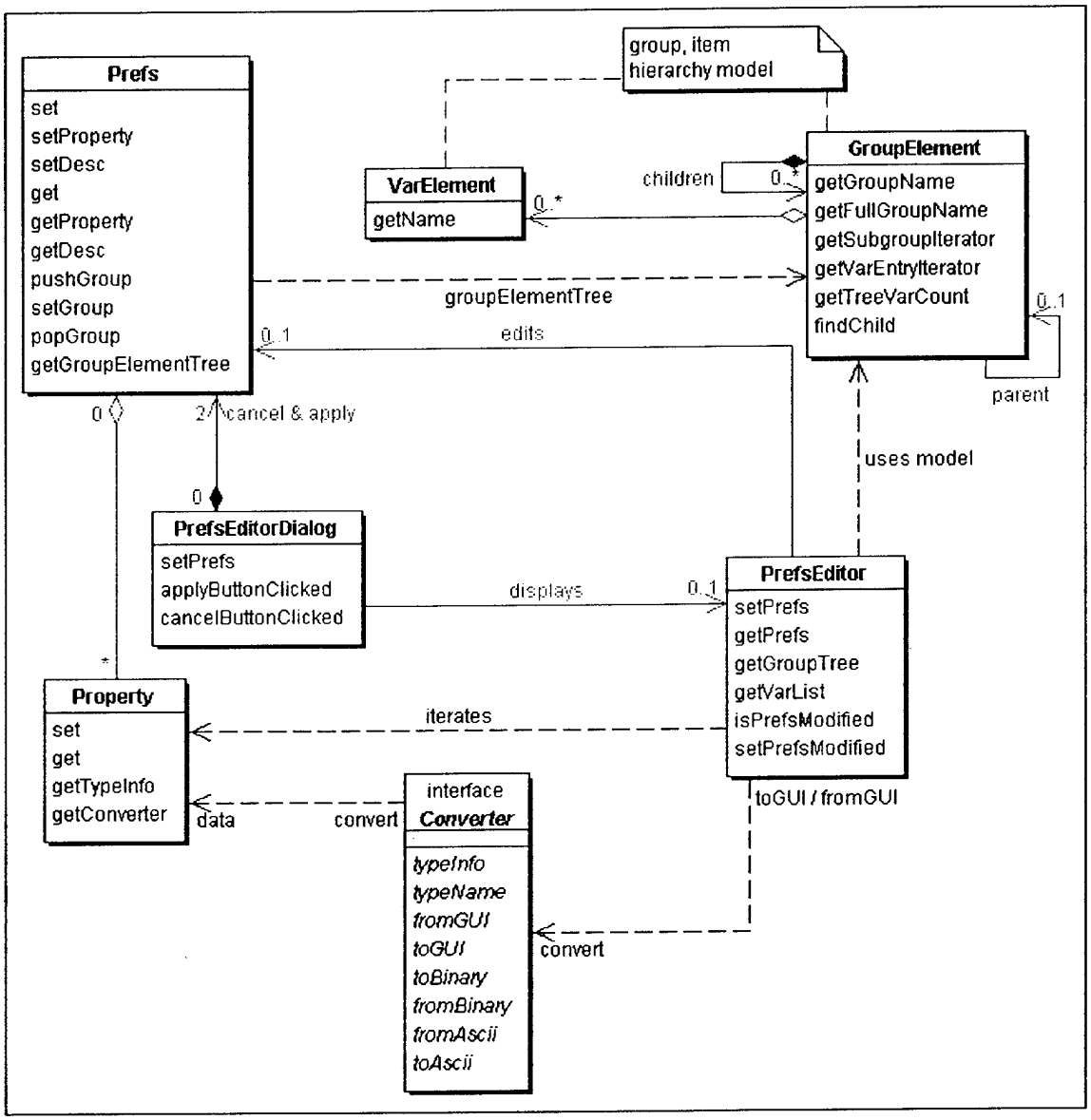

Figure 4: Preference Editor Pattern

Participant Descriptions

The following participants provide the interface or API to the users of the library.

Prefs

- Provides an interface to users of the preference system to store and retrieve all preferences.
- Central storage for the Property class.
- Categorizes each Property by using a variable name and group name.
- Provides a group name stack for setting and getting variables.
- Returns a tree using GroupElements and VarElements to browse the Prefs structure.

PrefsSerializer

- Interface for loading and saving a Prefs class.
- This class should be subclassed to support a particular format.
- The subclasses of PrefsSerializer must ensure groups, variables and values are saved and reloaded correctly.

PrefsSerializerRegistry

- Registers an implementation of PrefsSerializer.
- The PrefsSerializerRegistry is associated with a name defined by each PrefsSerializer.

PrefsDialog and **PrefsEditor**

- PrefsEditor provides a browsing and editing capability for the preferences.
- PrefsDialog displays the PrefsEditor in a window. PrefsDialog handles apply and cancel cases.
- Both the PrefsEditor and the PrefsDialog will require integration into the target platform toolkit.

The following participants provide internal functionality for the preference system.

Property

- Stores a single preference data item along with type information about that data item.
- Provides a convenience method to return a Converter for the Property type.

Converter

- Interface for converting a Property between the mediums: GUI, text, and binary.

- There must be an implementation of Converter for every type that the preference system needs to handle.

ConverterWidget

- A GUI component allows the user to edit a specific Property type.
- There must be an implementation of ConverterWidget for every implementation of Converter.

ConverterRegistry

- Associates the Converter with a particular data type.

GroupElement and **VarElement**

- Used to access the tree categorization of group and variable names defined by the Prefs class. The GroupElement class stores a group name. The VarElement class stores a variable name.
- Useful for operations such as creation of a GUI and serialization.
- The GroupElement and VarElement do not actually contain any preference values; they contain the group names and variable names.

Participant Collaborations

This section describes on a conceptual level how the participants work together. *Section 3: C++ Implementation of Preference Pattern* discusses several implementation issues involved in the actual implementation of several of these classes.

Prefs

- set(), get(), getProperty()
  - Uses Property to store and retrieve the values.
- getGroupElementTree()
  - Creates the GroupElement and VarElements.
  - Returns parent GroupElement.
- pushGroup(), popGroup(), getGroup()
  - Group stack accessor methods are provided as convenience instead of passing the group information to every set() and get() method call.
  - These methods allow the group name to be hidden from the code where individual variables are set. This allows for greater flexibility and maintainability of group categorization.

GroupElement and **VarElement**

- Uses a tree structure where the first GroupElement is the parent node and the VarElements are the children nodes.
- A GroupElement can contain another GroupElement and VarElement as children.

ConverterRegistry

- Uses the type information method defined by the Converter interface to register the Converter.
- When a Converter is needed for a Property, it can be requested from the ConverterRegistry by matching the registered type information with the Property type information.

Subclasses of Converter

- Converts between a medium and the corresponding Property type. The mediums currently defined for conversion are text, binary, and ConverterWidget.
- Uses the corresponding ConverterWidget to convert between the GUI.

Property

- getConverter()
  - Passes its own type information to the ConverterRegistry which returns a Converter for the Property.

PrefsEditor

- The editor uses a layout scheme that allows the user to browse the group and variables. The editor also must allow the user to edit the values for the variables.
- The browser is built by using the structure provided by using the GroupElement and VarElement classes. The GroupElement is retrieved by calling the Prefs method getElementGroupTree(). The model tree provides the group organizations and the variable names to layout the editor.
- The ConverterWidget must be retrieved for the variable to actually edit the object. Once the editor widget is retrieved, it must be added to the layout of the PrefsEditor.
  - To retrieve the ConverterWidget, the variable name must be passed to the Prefs class. The variable names will be known from the model tree.

The Prefs class returns a Property which provides the corresponding ConverterWidget. The actual method calls will be similar to:

```
// get the property given the variable and group name
Property &prop = prefs.getProperty(varName, groupName);

// get the converter widget from the property and show it
ConverterWidget widget = prop.getConverter().toGUI();

addWidgetToLayout(widget);
```

- After the ConverterWidget has been modified, the value needs to be extracted out of the ConverterWidget and stored back into the Prefs class. The method calls to perform that action should be similar to:

```
// get the property back out of the widget
Property &prop = property.getConverter().fromGUI( widget );

// store the modified property into the Prefs object
prefs.set(varName, prop, groupName);
```

- The editor will need to be rebuilt or relayed out whenever the setPrefs() method is called.

PrefsDialog

- Composed of a PrefsEditor.
- If apply is clicked, the Prefs object used in the PrefsEditor is copied to the Prefs object passed to the dialog.
  - If cancel is clicked, no reassignments are done to the original Prefs object passed to the dialog.

Subclasses of PrefsSerializer

- The save() method uses the model provided by Prefs getGroupElementTree() to traverse through all preference elements.
- For every VarElement, saves the GroupElement full group name. For each VarElement it saves the variable name and corresponding Property value.
- Individual preference values can be saved to text files using property.getConveter().toAscii() methods. Binary files can use the toBinary() methods.

- The load() method should read and tokenize each line.
  - Each line should contain the variable name, group, description and value.
  - These values should be passed to the Prefs class set() method.

PrefsSerializerRegistry

- The library user must register a subclass of PrefsSerializer and associate with a meaningful name

Consequences

The preference pattern has the following consequences

1. Guarantees data will be type safely stored and retrieved.
2. Property class and the implementations of Converter decouple the original data item from how to edit and save the object.
3. Conversion algorithms between strings, binary, and GUI can be changed easily without other operations such as saving and loading being aware.
4. By using multiple ConverterRegistry objects, multiple conversion schemes can be used. For example, a SQL ConverterRegistry can be used to convert information from a database. A different ConverterRegistry can be used to convert the same data type from a user specified format.
5. Every type that needs to be converted (i.e. any type that is a preference) must have a Converter defined for it. This could lead to cases where the developer creates a Prefs object, storing a type that the Converter cannot handle, preventing the user from entering that information and saving or loading that Property.

Pattern Remarks

The preference pattern itself is actually a composite of relatively straightforward predefined patterns[3]. The GroupElement and VarElement is actually just a variation of the Composite pattern. The Property and Converter is a variation of the Bridge pattern. The ConverterRegistry and the PrefsSerializerRegistry is a Builder pattern.

[3] See Design Patterns: Elements of Object-Oriented Software for further definitions of the referred patterns.

Section 3: C++ Implementation of Preference Pattern

The implementation of the preference pattern has some difficult issues to overcome. This section will outline many of the issues involved.

The core and perhaps hardest decision in implementing the preference pattern involves determining the type of an object at run time and general object anonymity. The other issues discussed in this section are: using the group/variable tree, creating a generic editor, deciding between multiple ConverterRegistry objects or a using singleton pattern, storing all necessary information for loading and saving, and defining all valid Converter types.

In the following list of possible C++ solutions to these problems are provided. A guide to help the developer choose among the proposed solutions is also provided when appropriate.

Determining the Type of an Object at Run Time

Issues

1. Matching the appropriate Converter to a Property
2. Type checking when storing or retrieving a value with a Property.

Solutions

Two possible approaches of determining the type of an object at run time are to use compiler supported run time type identification or using inheritance.

Run Time Type Identification

The Converter and Property classes can be implemented by utilizing compiler supported run time type identification (RTTI)[4]. The primary implementation issues are listed below[5].

1. The Property class's set() method should copy the type_info class and the value of the variable. The next subheading *Using Object Anonymity* discusses possible solutions to storing and retrieving the value of the variable.

---

[4] The C++ standard defines a method `typeid(<variable>)` that returns a `type_info` class. The `type_info` class has a method that will return the name (`const char *`) of the type.

[5] Please see Appendix B for the full implementation of the `Property` and `Converter` classes.

2. The Property class's get(&<dest var>) method should take the destination variable in which to store the Property's value. Then a type comparison can be performed between the destination variable and the stored type_info class. An exception can be thrown if the type match fails.
3. A Converter for each valid Property type should be registered to the ConverterRegistry. The ConverterRegistry would associate the name returned by the Converter's type_info with the Converter class. After a Converter is registered, a Property can pass the its type_info to the ConverterRegistry and the Property's associated Converter will be returned.

Benefits

The RTTI approach involves only a few lines of code and should have no impact on the design of the rest of the application.

Liabilities

The downside to this approach is the dependence that compiler will follow the C++ standards and implement RTTI. However, our organization is doing cross platform development using three different compilers and found RTTI to be implemented by all compilers.

Inheritance

Another option is to use inheritance to provide type information. Inheritance from the Property class must be used to provide an implementation for each preference type. The primary implementation steps are listed below.

1. Property base class is actually an interface that defines the set(), get()and a getTypeName() methods. These methods must be implemented in each subclass of Property.
2. Converter getTypeName() methods return value are matched against the Property getTypeName() method. The Property and Converter getTypeName() methods are used to match subclasses of Converter up with subclasses of Property.

Benefits

This solution guarantees compiler independence and provides a mechanism for object reflection similar to Java's.

Liabilities

The inheritance solution will have a large impact on the design of the Prefs system and possibly an application; i.e. possible force future use of multiple

inheritance. Also, there would be an extremely large overhead for having to write a Property class for every type.

The user would have to instantiate the appropriate Property upon creation of a class instead of being able to call a set method to an already existing Property object.

There would be no ability to perform Property assignments since each Property may be of a different type. For instance, you could not assign a DoubleProperty to a IntProperty without writing overloaded equal operators between every type.

Using Object Anonymity in Property Implementation

Issues

1. Storing any preference data object inside of the Property class.
2. Passing any preference data type to Property's set() method.
3. Retrieving the preference data back out of the Property class.
4. Deep copy versus shallow copy when assigning an object to the Property class.

Solutions

Two approaches are presented to solve the object anonymity problem. One is to use templated member functions and a templated storage class. The other is to use the same inheritance solution described in the subsection *Determining the Type of an Object at Run Time.*

Either solution will eventually involve the Property class storing a data object. The third subsection describes the benefits an liabilities of performing a deep or shallow copy of the data object.

Templated Member Functions and Templated Storage Class

The Property class's set() and get() methods can be implemented using templated member functions to pass and return the values in and out of the class.

Templated member functions allow the methods to be templated while the class itself does not have to be templated. A non-templated class is important since later when referring the class during operations such as saving or

converting to the GUI, each Property's type is unknown; i.e. the type to reference the Property will be unknown.

Templated member functions allow data to be generically passed into the Property class, however the issue of how to store this templated data as a attribute of the Property class needs to be resolved. The two approaches to storing the templated data is described below.

1. The simple but flawed approach is to store the data using a void pointer by doing an assignment such as:
   fData = static_cast<void *>(copiedTemplatedParam);
   The problem with this approach is in the destruction of the data. Since the data is being stored as a void pointer, the compiler does not know what type the value is and is unable to properly call the destructor of the type and free the memory. The memory leak problem will only occur if the Property class is doing a deep copy (recommended) instead of a shallow copy.
2. A slightly more complex solution but one that does not leak memory is to create a base class called DataHolder, which provides an interface to retrieve the data from the internal class. The DataHolder can be subclassed to a TypedDataHolder. The TypedDataHolder is actually a templated class and stores the data using the templated type as the data as shown from the code excerpt from Appendix A.03 – Property.h below:

```
class DataHolder
  {
  public:
    virtual ~DataHolder() { }

    virtual void get(void *data, const std::type_info& tinfo) const=0;

    virtual const std::type_info &typeInfo()const=0;
    virtual DataHolder *clone() const=0;
  };

template <class type>
class TypedDataHolder : public DataHolder
  {
  public :
    TypedDataHolder(const type data, const std::type_info& tinfo) :
          fData(data), fTypeInfo(tinfo)
          {
          if (typeid(fData) != tinfo)
```

```
                throw PropertyTypeMismatchException("TypedDataHolder(data, info)", fTypeInfo.name(),
                                                                    tinfo.name());
        }

    ~TypedDataHolder()
        { }

    void get(void *data, const std::type_info& tinfo) const
        {
        if (tinfo != fTypeInfo)
            throw PropertyTypeMismatchException("get", fTypeInfo.name(), tinfo.name());

        type *typedParam = (type *)data;
        *typedParam = fData;
        }

    const std::type_info &typeInfo() const { return fTypeInfo; }

    DataHolder *clone() const
        {
        return new TypedDataHolder<type>(fData, fTypeInfo);
        }

  private :
    type fData;
    const std::type_info& fTypeInfo;
};
```

Examples

We defined the Property get() method to take both the destination variable and the type_info as parameters. This forces the caller of the Property object to explicitly specify what type of object they think they should be retrieving which adds to code readability and maintainability.

The following example shows how to use the preference library to store and retrieve information in and out of a Prefs object. Internally, these methods will use a Property object to store or retrieve the information.

```
Prefs prefs;
prefs.set("maxWidth", 1000, type_id(int), "Windows", "Maximum width of a
                                                        window");
prefs.set("userName", QString(""), type_id(QString), "Personal", "User name");
...
int width;
QString userName;
```

```
prefs.get("maxWidth", width, type_id(int), "Windows");
prefs.get("userName", userName, type_id(QString), "Personal");
```

Inheritance

The Property class can also be implemented by using inheritance as described in the subsection above: *Determining the Type of an Object at Run Time*. The same benefits and liabilities for RTTI versus inheritance described in *Determining the Type of an Object at Run Time* applies for Object Anonymity as well.

Deep or Shallow Copy

The second issue to be determined with either solution is whether a deep copy or shallow copy is done in the set() method.

A deep copy is much safer by guaranteeing no dangling pointer and ensures data encapsulation. A deep copy however requires that every class that needs to be stored into the Property have an overloaded equals operator or copy constructor defined for that type. This shouldn't be a problem since all preference types should be a relatively basic type and be easily and obviously assignable.

A shallow copy could allow for the possibility of dangling pointers and breaks encapsulation since a external code could modify the pointer. A shallow copy would have a faster performance since a copy would not need to be done during preference assignment and retrieval. Regardless, breaking encapsulation is a major object oriented mistake and should be avoided.

Using the Group/Variable Tree

Issues

1. Browsing and iterating through the structure of the preferences.
2. Implementation of PrefsSerializer.
3. Implementation of PrefsEditor.

Solutions

Browsing and Iterating

The group/variable tree is represented by two classes, GroupElement and VarElement.

The GroupElement objects are the parent nodes while the VarElement objects are the children nodes. GroupElement satisfies the composite pattern by being composed of itself as well as the VarElement. At each level, group and variable names are guaranteed to be unique; i.e. two groups cannot have the same name and two variables cannot have the same name.

The group/variable tree only contains group names and variable names and does not contain the values of individual preferences. To retrieve values of individual preferences, the Prefs get() method must be called, passing the variable and group information provided by the model.

Iterating through the model should look similar to the following code. The example code outputs the group and variable tree plus the value of each preference.

```
Prefs prefs;
...
// preferences have already been stored
GroupElement *parent = prefs.getGroupElementTree();
recursiveOutput(prefs, parent);
...
void recursiveOutput(Prefs &prefs, GroupElement *node)
  {
 // output variable names for the group
  for (QListIterator<VarElement> entryIter =
          node->getVarEntryIterator();
          entryIter.current();
          ++entryIter)
      outputVarElement(prefs, *entryIter.current(), *node);

  // recursively iterate through the tree
  for (QListIterator<GroupElement> childrenIter =
          node->getSubgroupIterator();
          childrenIter.current();
          ++childrenIter)
      recursiveOutput(prefs, childrenIter.current();
  }

void outputVarElement(Prefs &prefs,
                      const VarElement &entry,
                      const GroupElement &group)
```

```
    {
    cout << group.getFullGroupName();
    cout << ">" << entry.getName();
    const Property &prop = prefs.getProperty(entry.getName(),
                                             group.getFullGroupName());
    cout << " = " << prop.getConverter().toAscii( prop );
  }
```

Serializer and Editor

A similar approach to the above example could be used in the implementation of the PrefsEditor or in subclasses of PrefsSerializer.

The saving algorithm would replace the functionoutputVarElement() with a function called saveElement(). The preference information can either be saved to a text or binary file. The use of the text conversion methods (toAscii(), fromAscii()) versus the binary conversion methods (toBinary(), fromBinary()) depends on whether the file format is text or binary.

As discussed in the *Participant Description* and *Participant Collaboration* subsections in Section 2, the PrefsEditor allows the user to browse and edit the preferences. The algorithm required to group and layout the browsing and editing components will need to iterate through the group and variables similar to the example above. The points below suggest possible alterations to the example in the building of the PrefsEditor layout:

- Instead of just iterating through the groups and variables, a tree GUI component node would be created and inserted with the group or variable name.
- The variable nodes (VarElement) could either be children of the group tree or could be a separated list displayed when a group is selected.

Creating a Standard Editor

Issues

1. Integrating the ConverterWidget, PrefsEditor, and PrefsDialog with the target platform or toolkit.
2. Organizing the PrefsEditor to provide the user with straightforward editing of the preferences.
3. Performing validity checking on data from the user.

Solutions

Please read the *Participant Descriptions* in Section 2 for a description of the PrefsEditor and PrefsDialog. This section's subsection *Using the Group/Variable tree* describes how to use the group/variable tree to get the necessary information for building the editor.

Editor Layout

The editor needs to present the user with the variable names, variable descriptions, ability to edit the values and some intelligent organization. Two possible approaches are to use a modified tree browser to edit one preference at time or to use a group pane dialog.

The modified tree browser creates an editor split into two panes. The left pane could be a tree browser as described in *Using the Group/Variable* tree. The right pane could be the editor for the selected variable in the tree. The right pane might also display the description for the editing variable. An example of this organization is shown below.

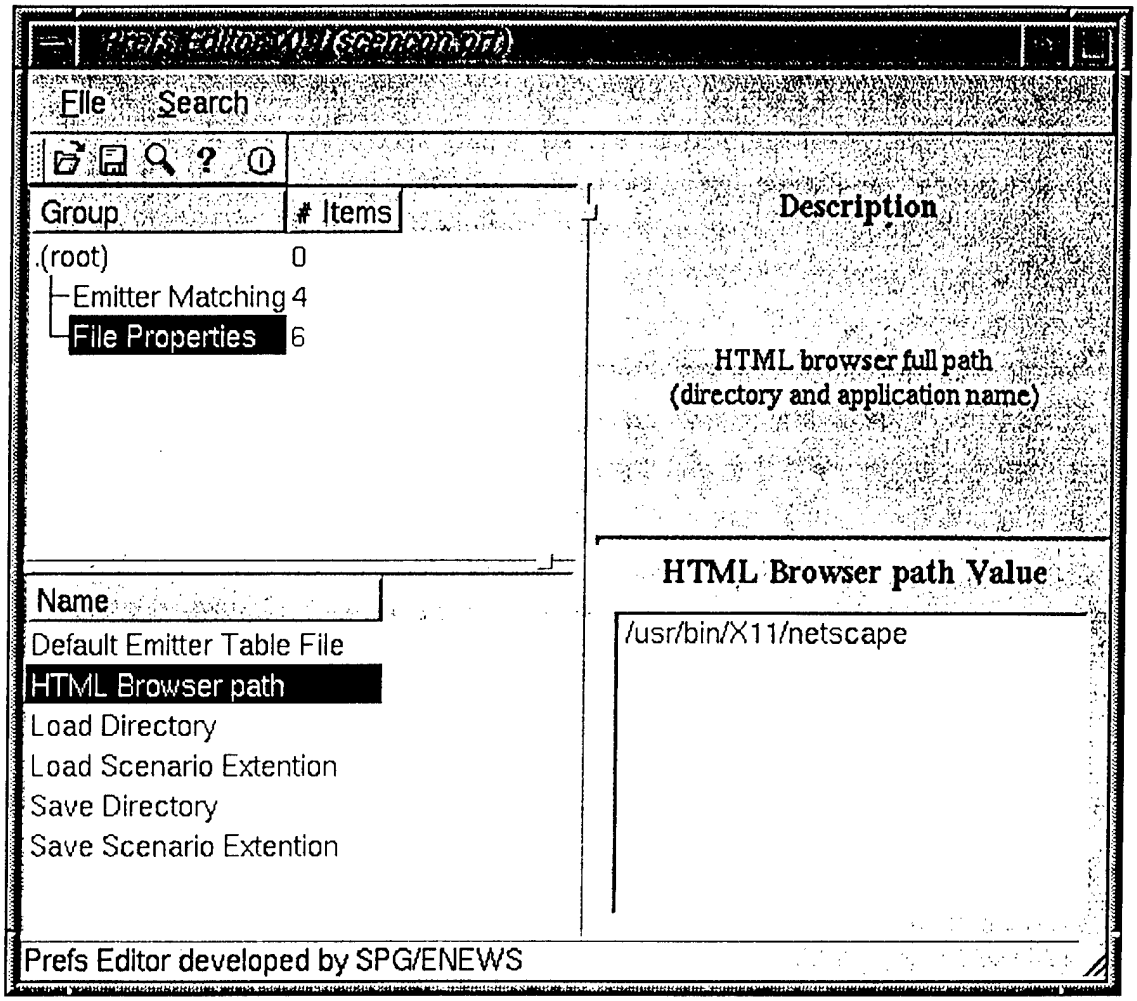

Figure 5: Split Pane Prefs Editor

Another approach is to use a tab dialog for the group names. Inside each tab would be a vertical list containing a line that contains the variable name, the variable descriptions and the editor widget for that variable.

GUI Integration

Any graphical component described in the pattern needs to be integrated into the target platform or GUI toolkit. The classes involved in integrate with the target windowing environment are PrefsEditor, PrefsDialog and ConverterWidget.

The PrefsEditor and ConverterWidget should inherit from the common graphical widget such as QWidget for Qt or JComponent for Java 1.2. The PrefsDialog should inherit from a window or dialog component such as QDialog for Qt. The subclasses of ConverterWidget should probably be composed of basic

graphical editing components such as QStringConverter being composed of QMultiLineEdit widget for Qt.

Validity Checking

Each implementation of each ConverterWidget's getValue() method should perform some validity checking on the data received from the user. Before storing the value from the input component into a Property, the value should validated to be of the correct type. For example, a double value could not be stored into an integer Property. Ideally, the ConverterWidget should be implemented so only the valid information for that type could be entered by the user.

Multiple ConverterRegistry Instances or Singleton

Issues

1. Should multiple conversion formats for Property objects be allowed within one application.

Solutions

It is helpful to recognize whether multiple conversion formats will be used within one application or whether it is virtually guaranteed to use only one conversion algorithm. For example, there may be an application where Property object will be saved to both an SQL database, which has one conversion format style, and a set of Property objects that will be saved to a XML preference file. If it is not clear that only a single registry will be used, it is a good design decision to plan on using multiple ConverterRegistry objects.

- If multiple conversions are required, then the ConverterRegistry should be passed through to the Property when Property.getConverter() is called.
- If multiple conversions are not required, then Property.getConverter() can return the appropriate Converter by using the ConverterRegistry.instance() singleton method.

Saving and Loading file format

Issues

1. Format must support storing group information.
2. Format must support storing type information that is valid across compilers.
3. Binary versus text formats.

Solutions

There are no hard and fast solutions to the issues listed above, but they should be kept in mind when creating a file format.

Whatever the file format that is chosen, make sure it does not limit the naming of a group or variable name. For example, the file format probably should not delimit on spaces or quotes since both will probably be used inside variable or group names. Possible solutions to choosing an serialization implementation are listed below.

- One possible solution is to create a simple text preference file format as we did in our solution.
- Another more flexible solution may be to use XML to define the preference file format.
- Saving in a binary file is also a possibility; however saving in a text file still allows the user to be able to edit the file manually. This maybe considered a benefit or liability depending on your situation.

One possible source of complications may come about in storing the type information. If Property is implemented using RTTI, the type name string should not be used since the string returned is not the same across platforms or even the same across compilers.

A solution is to create a method in the Converter called getTypeName() which must be implemented in the subclass of each Converter. This guarantees that the type name will be valid across compilers and across platforms. This API is shown in the structure diagrams of the pattern.

Documenting All Valid Types

Issues

1. Documenting a rule for valid preference types.
2. Creating a the appropriate subclasses for all valid types.

Solutions

Rules for Valid Preference Types

It is a good idea to define a set of valid preference types to provide discipline to the users of the preference system. For example a class that contains lists, strings and booleans probably is too complex to be considered a preference type. Here are possible standards that implementers of the preference system may want to adopt.

- Support only built in types (such as int, float, double, etc.). This solution will require the smallest amount of coding in the implementation of the Converter classes. However this is probably not sufficient for most applications.
- Support built in types plus a small-defined set of simple classes such as Color , Date, and String.
- Support the types above plus a list of the type basic types.

Creating Subclasses for All Valid Types

If using RTTI, a subclass for Converter and ConverterWidget should be defined for each valid preference type. If not using RTTI, then a subclass for Property must also exist for each valid preference type.

Section 4: Conclusion

The preference pattern and C++ implementation solves the common requirement to provide a user interface and serialization ability for a generic set of preferences. This report outlined the preference pattern's applicability, usability, structure, participants and participant collaboration. The C++ implementation section described the key implementation issues of the pattern and some possible solution to those issues.

The preference system provides a method of being able to convert a set of generic properties to multiple mediums. For these set of requirements, the mediums were converting between a GUI and converting between a file. This same property conversion pattern may be extended for future projects where new conversion mediums may be added.

Section 5: References

Materials

Blaha, Stephen. *C++ for Professional Programmers.* International Thomson Computer Press, Boston, MA, 1995.

Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

Niemeyer, Peck. *Exploring Java, Second Edition.* O'Reilly & Associates, Sebastopol, CA, 1997.

SPG Developers and Researchers

- Gregory Stern
- Michael Pilone
- Daniel Pilone
- Brian Solan
- Lawrence Schuette
- James Durbin
- Brian Calves
- Thomas Diepenbrock

Appendix A – Sample C++ Code

- There is too much code to provide for a complete code listing in this report. The entire library can be requested by contacting the Naval Research Library in Washington DC. Please email all requests to stern@enews.nrl.navy.mil.
- The code listing given below provides the most difficult parts of the implementation including the Property class, the Converter interface, some sample converters and the SimplePrefsSerializer. Note that the implementation uses Qt's foundation libraries for data structures and graphical items. A non-commercial use of Qt can be downloaded for free at www.troll.no.

Appendix Table of Contents

Appendix A.01 - Prefs.h

```
#ifndef CPL_PREFS_H
#define CPL_PREFS_H
#include <qstring.h>
#include <qlist.h>
#include <qstringlist.h>
#include <qdict.h>
#include "CPL/Convert/Property.h"
#include "CPL/Prefs/GroupElement.h"
#include "PrefsException.h"

/*****************************************************************************
 *****                                                   *****
 *****              Classification: UNCLASSIFIED                 *****
 *****               Classified By:                          *****
 *****               Declassify On:                           *****
 *****                                                   *****
 *****************************************************************************
 *
 *
 * Developed by: Naval Reseapp Laboratory, Tactical Electronic Warfare Div.
 *          Effectiveness of Navy Electronic Warfare Systems, Code 5707
 *          4555 Overlook Ave.
 *          Washington, D.C. 20375-5339
 *
 * For more information please call 202-767-2897 or send email to
 * enews@enews.nrl.navy.mil
 *
 * The U.S. Government retains all rights to use, duplicate, distribute,
 * disclose, or release this software.
 */

/** class Prefs
  *
  * Base class for all configuration storage to map group
  * (category) names to
  * variable names and values.  The keys may be case sensitive.
  *
  * Each set function can take an optional group.  If group
  * string is not given, then the classes's current group string
  * (set by set method, or push calls) will be used.
  *
  * Each get method will throw
  *
```

```
 * Stores type information for each variable to allow for strict
 * type checking when saving or loading.
 *
 * The group string is separated by dots.  For example:
 * builder.graphics.gl.color is a valid group
*/

class Prefs
  {
  friend class PrefsSerialization;

  public:
   /**@param primeHashTableSize the number of buckets (prime number) to create the
        * collision hash table; see QDict html page for a list of primes
    */
   Prefs(bool keyCaseSensitive=false, int primeHashTableSize=521);


   /**@return true if the group.varName is found, ie. is gettable */
   bool exists(const QString &varName, QString group=QString::null) const;


   template<class type>
   bool set(const QString &varName, const type& value, const std::type_info& typeInfo,
            QString group=QString::null, QString desc=QString::null)
                {
                QString key = _getKey(group, varName);
                PrefsElement* aValuePtr = fEntryDict.find(key);
                if (aValuePtr == 0L)
                  {
                  try
                        {
                        Property* prop = new Property(value, typeInfo);
                        aValuePtr = new PrefsElement(varName, prop, desc);
                        fEntryDict.insert( key, aValuePtr);
                        }
                  catch (Exception e)
                        {
                        qDebug("Prefs::set: %s", e.toString().ascii());
                        return false;
                        }
                  }
                else
                  {
                  try
                        {
                        Property* prop = new Property(value, typeInfo);
                        aValuePtr->setProperty(prop);
```

```
                }
            catch (Exception e)
                {
                qDebug("Prefs::set: %s", e.toString().ascii());
                return false;
                }
            }
        return true;
        }

    bool setProperty(const QString &varName, Property* value,
                QString group=QString::null, QString desc=QString::null);

    /** Sets the description of a entry. If the entry exists, the desc will be set and
     * true will be returned. If the entry doesn't exist, false will be returned
     */
    bool setDesc(const QString &varName, const QString& desc, QString group=QString::null);

    template<class type>
    void get(const QString &varName, type& outValue, const std::type_info& typeInfo, QString group=QString::null)
        {
        QString key = _getKey(group, varName);
        PrefsElement *aValuePtr = fEntryDict.find( key );
        if (aValuePtr == 0L)
          throw PrefsEntryNotFoundException( key );

        Property* prop = &aValuePtr->getProperty();
        prop->get(outValue, typeInfo);
        }

    template<class type>
    void get(const QString &varName, type& outValue, const std::type_info& typeInfo, QString group, QString&
outDesc)
        {
        QString key = _getKey(group, varName);
        PrefsElement *aValuePtr = fEntryDict.find( key );
        if (aValuePtr == 0L)
          throw EntryNotFoundException( key );

        Property* prop = &aValuePtr->getProperty();
        outDesc = aValuePtr->getDesc();
        prop->get(outValue, typeInfo);
        }

    const Property& getProperty(const QString &varName, QString group=QString::null) const;
    const QString getDesc(const QString &varName, QString group=QString::null) const;

    /*****************************************/
```

```
    /** pushes another layer onto the group stack */
    void pushGroup          (const QString &aGroup);

    /** sets the current group string; seperate group/subgroup categories by using a period; such as
builder.graphics.colors */
    void setGroup (const QString &formatedGroupString);

    /**@return the popped group name*/
    QString popGroup        ();
    void popAllGroups       ();

    /** the current group stack string (. delimited); will be used if no group string is provided in the get/set methods;
          uses static method createGroupVarString( ) */
    QString getCurrentGroup          () const;

    /**@return a newed GroupPrefs that provides a model for the group hierarchy; user must delete the newed
GroupPrefs*/
    GroupElement *getGroupElementTree() const;

    bool isKeyCaseSensitive() const { return fCaseSensitive; }

    /**@param formatedGroupString is a . delimited String, containing a group tree
          @param lastTokenIsVar true (default) if the last token in the formatedGroupString is a variable name, not
group
          @param varName return parameter to store the var name if lastTokenIsVar is true
          @return a string list of group names and var name into the param if flagged */
    static QStringList parseGroupVarString(const QString &formatedGroupString, bool lastTokenIsVar=true,
QString *varName=0L);
    /**@return a single string using the . delimiter in between strings */
    static QString createGroupVarString(const QStringList &strList);

  protected:
    class PrefsElement
          {
           friend class Prefs;
           friend class QDict<PrefsElement>;

          public:
           Property& getProperty() const { return *fProperty; }
           QString getDesc() const { return fDesc; }
           QString getName() const { return fName; }

           void setName(const QString& name) { fName = name; }
           void setDesc(const QString& desc) { fDesc = desc; }
           void setProperty(Property* prop)  { delete(fProperty); fProperty = prop; }
```

```
        protected:
          PrefsElement(QString name, Property* property, QString desc)
                : fProperty( property ), fName( name ), fDesc( desc ) {}

          ~PrefsElement() { delete (fProperty); }

        private:
          Property* fProperty;
          QString fDesc;
          QString fName;
        };

  private:
    /**@shapeType DependencyLink
     * @label creates*/
    /*#  GroupPrefs lnkUnnamed2*/
    /**@shapeType DependencyLink
     * @label throws*/
    /*#  PrefsEntryNotFoundException lnkUnnamed1*/
    /**@shapeType DependencyLink
     * @label throws*/
    /*#  PropertyInvalidTypeException lnkUnnamed*/
    QString _getKey(QString group, const QString &varName) const;

    /** @supplierCardinality 0..*
     * @byValue*/
    QDict<PrefsElement> fEntryDict;

    QStringList fCurrentGroup;
    bool fCaseSensitive;
  };

#endif //CPL_PREFS_H
```

Appendix A.02 - Prefs.cpp

```
#include "Prefs.h"
#include "CPL/Convert/Property.h"
#include "CPL/Util/StringTokenizer.h"

Prefs::Prefs(bool caseSensitive, int hashTableSize) :fEntryDict(hashTableSize, caseSensitive),
fCaseSensitive(caseSensitive)
  {
  fEntryDict.setAutoDelete(true);
  }

QString Prefs::_getKey(QString group, const QString &varName) const
  {
```

```
    if (group.isNull())
            group = getCurrentGroup();
    QString key = group;
    if (!key.isEmpty())
            key += ".";
    key += varName;
//    const char *asciiKey = key.ascii(); // for inspection in cvd DEBUG
    return key;
    }

bool Prefs::setProperty(const QString &varName, Property* value,
                                QString group, QString desc)
    {
    if (value == 0L)
            return false;

    QString key = _getKey(group, varName);
    PrefsElement* aValuePtr = fEntryDict.find(key);
    if (aValuePtr == 0L)
            {
             aValuePtr = new PrefsElement(varName, value, desc);
             fEntryDict.insert( key, aValuePtr);
            }
    else
            {
             aValuePtr->setProperty(value);
            }

    return true;
    }

bool Prefs::setDesc(const QString &varName, const QString& desc, QString group)
    {
    QString key = _getKey(group, varName);
    PrefsElement* aValuePtr = fEntryDict.find(key);
    if (aValuePtr == 0L)
            {
            return false;
            }
    else
            {
             aValuePtr->setDesc(desc);
            }

    return true;
    }

const Property& Prefs::getProperty(const QString &varName, QString group) const
```

```
    {
    QString key = _getKey(group, varName);
    PrefsElement *aValuePtr = fEntryDict.find( key );
    if (aValuePtr == 0L)
            throw PrefsEntryNotFoundException( key );

    return aValuePtr->getProperty();
    }

const QString Prefs::getDesc(const QString &varName, QString group) const
    {
    QString key = _getKey(group, varName);
    PrefsElement *aValuePtr = fEntryDict.find( key );
    if (aValuePtr == 0L)
            throw PrefsEntryNotFoundException( key );

    return aValuePtr->getDesc();
    }

void Prefs::pushGroup(const QString &aGroup)
    {
    fCurrentGroup += aGroup;
    }

void Prefs::setGroup(const QString &formatedGroupString)
    {
    StringTokenizer t(formatedGroupString, ".");
    fCurrentGroup = t.toStringList();
    }
QString Prefs::popGroup()
    {
    QString lastGroup = fCurrentGroup.last();
    fCurrentGroup.remove( fCurrentGroup.fromLast() );
    return lastGroup;
    }
void Prefs::popAllGroups()
    {
    fCurrentGroup.clear();
    }

GroupElement *Prefs::getGroupElementTree() const
    {
    //qDebug("\nPrefs::getGroupElementTree()");
    GroupElement *root = new GroupElement("", 0L);
    QDictIterator<PrefsElement> dictIter (fEntryDict);

    PrefsElement *curPrefsElement;
    VarElement* aVarElement;
```

```
    GroupElement *insertInto;
    QStringList groupList;
    QString varName="";
    for (dictIter.toFirst();dictIter.current();++dictIter)
            {
            curPrefsElement = dictIter.current();
            //qDebug("\ncreating entry for key : '%s'", dictIter.currentKey().ascii());

            // find which group to insert the var entry, first check root
            if (dictIter.currentKey().isEmpty())
              insertInto = root;
            else // check existing subgroups
              {
              groupList = parseGroupVarString(dictIter.currentKey(), true, &varName);
              if ((insertInto = root->findChild( groupList )) == 0L)
                    {
                    //qDebug("\n  creating a new group sub tree");
                    // create a new group
                    insertInto = root->_createGroupSubTree( groupList );
                    }
              }

            // insert the var entry into the existing or created group
            aVarElement = new VarElement( curPrefsElement->getName() );
            insertInto->_addVarElement ( aVarElement );
            }
    return root;
    }

QString Prefs::createGroupVarString(const QStringList &strList)
    {
    QString g = "";
    QStringList::ConstIterator iter = strList.begin();
    for (int i=0;iter != strList.end();++iter, i++)
            {
            if (i > 0)
              g+= ".";
            g += *iter;
            }
    return g;
    }

QString Prefs::getCurrentGroup() const
    {
    return createGroupVarString(fCurrentGroup);
    }

bool Prefs::exists(const QString &varName, QString group) const
```

```
  {
  QString key = _getKey(group, varName);
  PrefsElement *aValuePtr = fEntryDict.find( key );
  return (aValuePtr != 0L);
  }

QStringList Prefs::parseGroupVarString(const QString &formatedGroupString, bool lastTokenIsVar, QString
*varName)
  {
  QStringList groupList;
  StringTokenizer tokenizer(formatedGroupString, ".");
  TokenIterator tokenIter(tokenizer);

  int lastToken = tokenizer.numTokens()-1;
  if (!lastTokenIsVar)
        lastToken++;
  for (;tokenIter.getCurTokenNum() < lastToken;tokenIter++)
        groupList += *tokenIter;
  bool canStore = (varName != 0L);
  if (canStore && (tokenIter.isValid()))
        *varName = *tokenIter;
  return groupList;
  }
```

Appendix A.03 - Property.h

```
#ifndef CPL_PROPERTY_H
#define CPL_PROPERTY_H

/*****************************************************************************
 *****                                                  *****
 *****            Classification: UNCLASSIFIED                *****
 *****             Classified By:                          *****
 *****             Declassify On:                          *****
 *****                                                  *****
 *****************************************************************************
 *
 *
 * Developed by: Naval Reseapp Laboratory, Tactical Electronic Warfare Div.
 *          Effectiveness of Navy Electronic Warfare Systems, Code 5707
 *          4555 Overlook Ave.
 *          Washington, D.C. 20375-5339
 *
 * For more information please call 202-767-2897 or send email to
 * enews@enews.nrl.navy.mil
 *
 * The U.S. Government retains all rights to use, duplicate, distribute,
 * disclose, or release this software.
 */
```

```
#include <qstring.h>
#include <qtextstream.h>
#include <qdatastream.h>

#include <typeinfo>

#include "ConvertException.h"
#include "CPL/Convert/Converter.h"
#include "CPL/Convert/ConverterFactory.h"
#include "CPL/CPLGlobal.h"
#include "Converter.h"

/**
 * This class is a simple templated class that uses rtti to allow storage of any
 * type, while at the same time maintaining type safety.
 *
 * For CORBA people: It is a poor mans CORBA::Any.
*/

////////////////////////////////////////////////////////////////////
// PropertyDataHolder  (internal class)                           //
////////////////////////////////////////////////////////////////////
class DataHolder
  {
  public:
    virtual ~DataHolder() { }

    virtual void get(void *data, const std::type_info& tinfo) const=0;

    virtual const std::type_info &typeInfo()const=0;
    virtual DataHolder *clone() const=0;
  };

template <class type>
class TypedDataHolder : public DataHolder
  {
  public :
    TypedDataHolder(const type data, const std::type_info& tinfo) :
            fData(data), fTypeInfo(tinfo)
            {
            if (typeid(fData) != tinfo)
                    throw PropertyTypeMismatchException("TypedDataHolder(data, info)", fTypeInfo.name(),
tinfo.name());
            }

    ~TypedDataHolder()
            {
```

```
            //cout << "deleting TypedDataHolder" << endl;
            }

    void get(void *data, const std::type_info& tinfo) const
            {
            if (tinfo != fTypeInfo)
                    throw PropertyTypeMismatchException("get", fTypeInfo.name(), tinfo.name());

            type *typedParam = (type *)data;
            *typedParam = fData;
            }

    const std::type_info &typeInfo() const { return fTypeInfo; }

    DataHolder *clone() const
            {
            return new TypedDataHolder<type>(fData, fTypeInfo);
            }

  private :
    type fData;
    const std::type_info& fTypeInfo;
  };

////////////////////////////////////////////////////////////////////
// Property                                                       //
////////////////////////////////////////////////////////////////////
  class Property
    {
    public:
      /** This constructor requires a value to ensure that the property is instantiated
       * correctly. Sets the value of this property. If an invalid type is given, this method
       * will throw a PropertyInvalidTypeException whcih will contain the names of the two
       * types that the property was trying to deal with. If an exception is thrown, the
  * Property will be left in an invalid state. It is recommended that you destroy the
  * Property in a catch statement.
       */
    template <class type>
    Property(const type t, const std::type_info& tinfo) : fDataHolder( 0L )
            {
            set(t, tinfo);
            }

    Property(const Property &copyFrom) : fDataHolder(0L)
            {
            DataHolder *copy = copyFrom.fDataHolder;
            if (copy)
                    copy = copy->clone();
```

```
        fDataHolder = copy;
        }

  /** Default constructor. Allows you create Property objects and fill the value in later
   * using the set method. If get is called before set, a PropertyEmptyException will
   * be thrown
   */
  Property() : fDataHolder( 0L )
        {}


  ~Property()
        {
        if (fDataHolder)
              delete fDataHolder;
        fDataHolder = 0L;
        }

/* Sets the value of this property. If an invalid type is given, this method will throw
   * PropertyInvalidTypeException which will contain the names of the two types that
   * the property was trying to deal with. If an exception is thrown, the Property will
   * be left in an invalid state. It is recommended that you destroy the property in a
* catch statement.
   */
  template <class type>
  void set(const type t, const std::type_info& tinfo)
        {
        if (tinfo != typeid(t))
              throw PropertyTypeMismatchException("set", typeid(t).name(), tinfo.name());

        delete fDataHolder;
        fDataHolder = (DataHolder *)new TypedDataHolder<type>(t, tinfo);
        }

  /** Gets the value back from the property. If an invalid type is given, this method
* will throw a PropertyTypeMismatchException which will contain the names of the two
* types that the property was trying to deal with. The original value should
* still be valid in the propertyeven if an exception is thrown
   */
  template <class type>
  void get(type& t, const std::type_info& tinfo) const
        {
        if (tinfo != typeid(t))
              throw PropertyTypeMismatchException("get", typeid(t).name(), tinfo.name());

        if (!fDataHolder)
              throw PropertyEmptyException("Propery::get");
```

```
        fDataHolder->get((void *)&t, tinfo);
        }


    virtual Property& operator=(const Property &copyFrom)
        {
        DataHolder *copy = copyFrom.fDataHolder;
        if (copy)
            copy = copy->clone();
        delete fDataHolder;

        fDataHolder = copy;
        return *this;
        }

    /** Returns the type info for this type. Will throw a PropertyEmptyException if there
     * is no type in this property
     */
    virtual const std::type_info& getTypeInfo()
        {
        if (!fDataHolder)
            throw PropertyEmptyException("Property::getTypeInfo");

        return fDataHolder->typeInfo();
        }

    /** Returns the converter for this property using the ConverterFactory
     */
    virtual Converter& getConverter() const
        {
        if (!fDataHolder)
            throw PropertyEmptyException("Property::getConverter");

        return ConverterFactory::instance().getConverter( fDataHolder->typeInfo() );
        }

    /** Returns the converter for this property using the given converter
     */
    virtual Converter& getConverter(AbstractConverterFactory& factory) const
        {
        if (!fDataHolder)
            throw PropertyEmptyException("Property::getConverter");

        return factory.getConverter( fDataHolder->typeInfo() );
        }

protected:
```

```
    DataHolder* fDataHolder;
  };

#endif
```

Appendix A.04 - Converter.h

```
#ifndef CPL_CONVERTER_H_
#define CPL_CONVERTER_H_

#include <qarray.h>
#include <qwidget.h>
#include <qstring.h>
#include <qsizepolicy.h>
#include <typeinfo>

#include "CPL/CPLGlobal.h"
#include "ConvertException.h"
#include "Property.h"

class ConverterWidget;

class Converter
  {
    friend class ConverterFactory;

  public:
    Converter() {};

    /** Takes an ascii ByteArray and parses it into something valid, then returns it in a
     * property. It is the callers job to free the property. Could throw a
* ConverterDataFormatException
     */
    virtual Property* fromAscii(const QString) = 0;

    /** Takes an binary ByteArray and parses it into something valid, then returns it in
     * a property. It is the callers job to free the property.  Could throw a
* ConverterDataFormatException
     */
    virtual Property* fromBinary(const QByteArray) = 0;

/** Takes a QWidget and uses the data in it to create a valid Property, returns the
 * property. It is the callers job to free the property and the QWidget after the
     * function call. Could throw a ConverterDataFormatException.
     */
    virtual Property* fromGUI(const ConverterWidget&) = 0;

/** Takes an ascii QString and parses it into something valid, then fills in the given
     * property. Could throw a ConverterDataFormatException
```

```
     */
    void fromAscii(const QString, Property& prop);

    /** Takes an binary ByteArray and parses it into something valid, then fills in the
     * given property. Could throw a ConverterDataFormatException
     */
    void fromBinary(const QByteArray, Property& prop);

    /** Takes a QWidget and uses the data in it to fill in the given property.
     * Could throw a ConverterDataFormatException
     */
    void fromGUI(const ConverterWidget&, Property& prop);

    /** Takes a property, encodes it, and returns the value in a ByteArray. to get
     * This method needs the value out of the property and therefore could trigger a
* PropertyTypeMismatchException
     */
    virtual QString toAscii(const Property&) const = 0;

/** Takes a property, encodes it, and returns the value in a ByteArray. This method
 * needs to get the value out of the property and therefore could trigger a
 * PropertyTypeMismatchException
      */
    virtual QByteArray toBinary(const Property&) const = 0;

    /** Takes a property, creates a widget, and returns the widget for use in a gui. This
* method needs to get the value out of the property and therefore could trigger a
     ** PropertyTypeMismatchException
     */
    virtual ConverterWidget* toGUI(const Property&, QWidget* parent = 0L, const char* name = 0L) const = 0;

    /** Returns the name of the type, like QInt32 or QInt16. The name of the converter
* should be exactly the same as the name for the class, minus the Converter. For
*  example: BoolConverter would have the name Bool
     */
    virtual QString typeName() const = 0;

    /** Returns the typeinfo for the type. This is platform dependent, so never
     * store this value past the end of the program execution or you could run into
     * problems if it is run on a different platform. Just use this method for simple
     * comparisons.
     */
    virtual const std::type_info& typeInfo() const = 0;

  protected:

  private:
```

```
    /** @link dependency */
    /*#  Property lnkProperty; */
};

class ConverterWidget : public QWidget
  {
   Q_OBJECT

  friend class Converter;
  public:
   ~ConverterWidget() {}

    /** Returns the typeinfo for the type. This is platform dependent, so never
     * store this value past the end of the program execution or you could run into
     * problems if it is run on a different platform. Just use this method for simple
     * comparisons.
     */
    virtual const std::type_info& typeInfo() const = 0;

     /** Sets the display of this widget to represent the given property. Could throw a
     * PropertyTypeMismatchException if the wrong type of property is passed in
     */
    virtual void setValue(const Property&) = 0;

    /** Gets the value out of the widget and into a property. It is the callers job
     * to free the property
     */
    virtual Property* getValue() const = 0;

    virtual QSizePolicy sizePolicy() const { return QSizePolicy( QSizePolicy::Preferred, QSizePolicy::Preferred ); }

  protected:
    ConverterWidget(QWidget* parent = 0L, const char* name = 0L)
            : QWidget(parent, name) {}
#define signals protected
  signals:
    void modified();

  protected slots :
    void slotModified(const QString&) { emit modified(); }
    void slotModified(int)          { emit modified(); }
    void slotModified()             { emit modified(); }
  };


#endif
```

Appendix A.05 - QInt32Converter.h

```
#ifndef CPL_QINT32_CONVERTER_H
#define CPL_QINT32_CONVERTER_H

#include "Converter.h"


class Property;
/** See Converter.h for documentation
 */
class QInt32Converter : public Converter
  {
  public:
    QInt32Converter();

    virtual QByteArray toBinary(const Property& property) const;
    virtual QString    toAscii(const Property& property) const;
    virtual ConverterWidget*   toGUI(const Property&, QWidget* parent = 0L, const char* name = 0L) const;
    virtual Property* fromAscii(const QString data);
    virtual Property* fromBinary(const QByteArray);
    virtual Property* fromGUI(const ConverterWidget&);

    virtual QString typeName() const            { return "QInt32"; }
    virtual const std::type_info& typeInfo() const { return typeid( Q_INT32 ); }

  };

class QLineEdit;
class CPL_EXPORT QInt32ConverterWidget : public ConverterWidget
  {
    friend class QInt32Converter;

  public:
    ~QInt32ConverterWidget() {}

    virtual const std::type_info& typeInfo() const { return typeid( Q_INT32 ); }

    void setValue(const Property&);
    Property* getValue() const;

  protected:
    QInt32ConverterWidget(const Property& property, QWidget* parent = 0L, const char* name = 0L);

    QLineEdit* fValueEdit;
  };
```

```
#endif
```

Appendix A.06 - QInt32Converter.cpp

```
#include "QInt32Converter.h"
#include "Property.h"

#include <qlineedit.h>
#include <qlayout.h>

#include <typeinfo>

QInt32Converter::QInt32Converter()
  {
  }

QByteArray QInt32Converter::toBinary(const Property& ) const
  {
  QByteArray buffer;

  qDebug("QInt32Converter::toBinary: Not implemented");

  return buffer;
  }

QString  QInt32Converter::toAscii(const Property& property) const
  {
  // First lets get the value from the property
  Q_INT32 value;
  property.get(value, typeInfo());

  return QString::number(value);;
  }

ConverterWidget* QInt32Converter::toGUI(const Property& property, QWidget* parent, const char* name) const
  {
  // Create the widget
  return new QInt32ConverterWidget(property, parent, name);
  }

Property* QInt32Converter::fromAscii(const QString data)
  {
  bool ok;
  Q_INT32 number = data.toInt(&ok);

  if (!ok)
        throw ConverterDataFormatException(typeName());
```

```
    return new Property(( Q_INT32 )number, typeid( Q_INT32 ));
    }

Property* QInt32Converter::fromBinary(const QByteArray data)
    {
    qDebug("QInt32Converter::fromBinary: Not implemented");

    return 0L;
    }

Property* QInt32Converter::fromGUI(const ConverterWidget& widget)
    {
    if (widget.typeInfo() != typeInfo())
            throw ConverterWidgetException( typeName() );

    return widget.getValue();
    }

////////////////////////////////////////////////
// QInt32 CONVERTER WIDGET
////////////////////////////////////////////////
QInt32ConverterWidget::QInt32ConverterWidget(const Property& property, QWidget* parent, const char* name)
     : ConverterWidget(parent, name)
    {
    QVBoxLayout* layout = new QVBoxLayout(this);

    fValueEdit = new QLineEdit(this, "fValueEdit");
    fValueEdit->setMinimumSize(125, fValueEdit->sizeHint().height());
    connect(fValueEdit, SIGNAL( textChanged(const QString&) ), SLOT( slotModified(const QString&) ));
    layout->addWidget( fValueEdit );

    setMinimumSize( fValueEdit->minimumSize() );

    setValue(property);
    }

void QInt32ConverterWidget::setValue(const Property& property)
    {
    fValueEdit->setText(ConverterFactory::instance().getConverter( typeInfo() ).toAscii(property));
    }

Property* QInt32ConverterWidget::getValue() const
    {
    return ConverterFactory::instance().getConverter( typeInfo() ).fromAscii( fValueEdit->text() );
    }
```

Appendix A.07 - QStringConverter.cpp

```
#include "QStringConverter.h"
#include "Property.h"

#include <qmultilineedit.h>
#include <qlayout.h>

#include <typeinfo>

QStringConverter::QStringConverter()
  {
  }

QByteArray QStringConverter::toBinary(const Property&) const
  {
  QByteArray buffer;

  qDebug("QStringConverter::toBinary: Not implemented");

  return buffer;
  }

QString  QStringConverter::toAscii(const Property& property) const
  {
  // First lets get the value from the property
  QString value;
  property.get(value, typeInfo());

  return value;
  }

ConverterWidget* QStringConverter::toGUI(const Property& property, QWidget* parent, const char* name) const
  {
  // Create the widget
  return new QStringConverterWidget(property, parent, name);
  }

Property* QStringConverter::fromAscii(const QString data)
  {
  Property *prop = new Property(data, typeInfo());

  return prop;
  }

Property* QStringConverter::fromBinary(const QByteArray data)
  {
  qDebug("QStringConverter::fromBinary: Not implemented");
```

```
    return 0L;
    }

Property* QStringConverter::fromGUI(const ConverterWidget& widget)
    {
    if (widget.typeInfo() != typeInfo())
            throw ConverterWidgetException( typeName() );

    return widget.getValue();
    }

//////////////////////////////////////////////
// QSTRING CONVERTER WIDGET
//////////////////////////////////////////////
QStringConverterWidget::QStringConverterWidget(const Property& property, QWidget* parent, const char* name)
     : ConverterWidget(parent, name)
    {
    QVBoxLayout* layout = new QVBoxLayout( this );
    fValueEdit = new QMultiLineEdit(this, "fValueEdit");
    fValueEdit->setMinimumSize(125, 50);
    connect(fValueEdit, SIGNAL( textChanged() ), SLOT( slotModified()));
    layout->addWidget( fValueEdit );

    setMinimumSize( fValueEdit->minimumSize() );

    setValue(property);
    }

void QStringConverterWidget::setValue(const Property& property)
    {
    fValueEdit->setText(ConverterFactory::instance().getConverter( typeInfo() ).toAscii(property));
    }

Property* QStringConverterWidget::getValue() const
    {
    return ConverterFactory::instance().getConverter( typeInfo() ).fromAscii( fValueEdit->text() );;
    }
```

Appendix A.08 - SimplePrefsSerializer.h

```
#ifndef CPL_SIMPLEPREFSSERIALIZER_H
#define  CPL_SIMPLEPREFSSERIALIZER_H
#include <qtextstream.h>
#include "PrefsSerializer.h"

/*****************************************************************************
 *****                                                                   *****
```

```
*****              Classification: UNCLASSIFIED              *****
*****              Classified By:                            *****
*****              Declassify On:                            *****
*****                                                        *****
********************************************************************************
*
* $Source: /enews/proj/spg/cvsroot/CPL/Prefs/SimplePrefsSerializer.h,v $
* $Author: mpilone $
* $Date: 2000/02/01 19:59:14 $
* $Id: SimplePrefsSerializer.h,v 1.1.1.1 2000/02/01 19:59:14 mpilone Exp $
*
* Developed by: Naval Research Laboratory, Tactical Electronic Warfare Div.
*           Effectiveness of Navy Electronic Warfare Systems, Code 5707
*           4555 Overlook Ave.
*           Washington, D.C. 20375-5339
*
* For more information please call 202-767-2897 or send email to
* enews@enews.nrl.navy.mil
*
* The U.S. Government retains all rights to use, duplicate, distribute,
* disclose, or release this software.
*/

/** simply loads and saves a prefs class to and from a text io device (such as File)
   will output debug statements if invalid type and shouldThrowType = false
*/
#include <qiodevice.h>

class VarElement;
class GroupElement;
class TokenIterator;
class Property;
class Converter;

/**
 * This class provides a simple serializer that will dump out plain text, typesafe
 * pref files. This class WILL NOT support multiline text. If you need to save multiline
 * text it is recommended that you use the XMLPrefsSerializer or a binary serializer
 * Caller must provide a prefs for this class to use, as well as a read/write iodevice. If the
 * io device is a file, it is recommended that the file be opened with the truncate option so
 * old data in the file is erased.
*/
class SimplePrefsSerializer : public PrefsSerializer
  {
  public:

    SimplePrefsSerializer();
```

```
    /** saves prefs into a io device
          throws IOException or ConverterUnsupportedException upon writing error */
    virtual void save(Prefs& prefs, QIODevice &writableDevice);

    /** parses the text io device for prefs and sets them into the current prefs;
          note that this will just overwrite any current prefs, but also keep any old prefs that
          aren't overwritten; user must reset the prefs prior to load for the prefs to contain
          only what is in the io device
    throws IOException or ConverterUnsupportedException upon reading error */
    virtual void load(Prefs& prefs, QIODevice &readableDevice);

    virtual QString typeName() const { return "Simple"; }

  private:
    void _saveEntry(Prefs& prefs, const VarElement &element, const GroupElement &group);
    void _recursiveSave(Prefs& prefs, GroupElement *node);

/** Checks for c/c++ style comments and returns true if the end of a comment was found or     * no comment
found, false otherwise. Continuation means that the line being passed in is
* still inside a comment that was started on a previous line. The caller can do something * like: <BR>
     * continuation = !_cutOutCommentArea(aLine, continuation);
     */
    bool _cutOutCommentArea(QString &line, bool continuation = false);

    /** parses the line and stores the information in the prefs class
          upon error, throws InvalidFormatException or UnrecognizedTypeException*/
    void _extractLine(Prefs& prefs, QString line, int lineNum);

/**@param tokenIter precondition: tokenIter at the group.varName pos; postcondition: valid iter at var name token
          extracts the group from the tokens, checking for correct format;
          upon error thros InvalidFormatException
          @return the group string */
    QString _extractGroup(TokenIterator &tokenIter, const QString &line, const int lineNum);

    /**@param tokenIter precondition: valid token at the name; postcondition: valid token at "<" delimtiter
          extracts the name from the tokens
          @return the name */
    QString _extractName(TokenIterator &tokenIter, const QString &line, const int lineNum);

    /**@param tokenIter precondition: iter at delimiter "<"; postcondition: valid iter one token past ">"
          extracts the type from the tokens, checking for correct format;
          upon error throws InvalidFormatException
          @return the property type */
    QString _extractType(TokenIterator &tokenIter, const QString &line, const int lineNum);

    /**@param tokenIter precondition: valid iter after delimiter ">"; postcondition: possibly
          invalid iter after value extracts the value from the tokens, checking for correct;
          format upon error throws InvalidFormatException
```

```
        @return the property type */
    Property* _extractValue(TokenIterator &tokenIter, Converter& converter, const QString &line, const int
lineNum);

    QString _extractDesc(TokenIterator &tokenIter, const QString &line, const int lineNum);

    /** all parameters must be in correct format and valid
        calls the appropriate set method in the fPrefs */
    void _setPrefs(Prefs& prefs, const QString &groupName, const QString &varName, Property* value);

    void _setPrefs(Prefs& prefs, const QString &groupName, const QString &varName, const QString &desc);

    QTextStream fTextStream;
    QTextStream fKeyValueStream;
    QTextStream fDescStream;
  };


#endif // CPL_SIMPLEPREFSSERIALIZER_H
```

Appendix A.09 - SimplePrefsSerializer.cpp

```
#include <qiodevice.h>
#include <qdatetime.h>
#include <qtextstream.h>
#include <qstringlist.h>
#include <qregexp.h>
#include <qbuffer.h>
#include "SimplePrefsSerializer.h"
#include "Prefs.h"
#include "VarElement.h"
#include "GroupElement.h"
#include "CPL/Convert/Property.h"
#include "CPL/Util/StringTokenizer.h"
#include "CPL/Util/UtilException.h"

SimplePrefsSerializer::SimplePrefsSerializer() :
    PrefsSerializer()
  {
  }

void SimplePrefsSerializer::save(Prefs& prefs, QIODevice &writableDevice)
  {
  if (!writableDevice.isWritable())
        throw IOException("Input/Output device is not writable");

  QTextStream outstream(&writableDevice);
```

```
    GroupElement *groupRoot = prefs.getGroupElementTree();

    outstream << _createFirstLineOfFile();
    outstream << "/*****************************************************************\n";
    outstream << " * Written by cpl/prefs/Prefs on " << QDate::currentDate().toString();
    outstream << " at " << QTime::currentTime().toString() << "\n";
    outstream << "*****************************************************************/";

    // Setup the test streams that will be used
    QBuffer *buffer = new QBuffer;
    buffer->open(IO_WriteOnly);
    fKeyValueStream.setDevice( buffer );
    buffer = new QBuffer;
    buffer->open(IO_WriteOnly);
    fDescStream.setDevice( buffer );

    _recursiveSave(prefs, groupRoot);


    outstream << "\n\n/**=-=-=-=-=-=-=-[ KEY / VALUE PAIRS ]=-=-=-=-=-=-=-**/";
    buffer = (QBuffer*)(fKeyValueStream.device());
    fKeyValueStream.unsetDevice();
    _checkStatus(outstream.device());
    outstream.writeRawBytes(buffer->buffer().data(), buffer->size());
    delete( buffer );

    outstream << "\n\n/**=-=-=-=-=-[ DESCRIPTIONS (DO NOT EDIT) ]-=-=-=-=-=-**/";
    buffer = (QBuffer*)(fDescStream.device());
    fDescStream.unsetDevice();
    _checkStatus(outstream.device());
    outstream.writeRawBytes(buffer->buffer().data(), buffer->size());
    delete( buffer );

    // check to make sure io device is still valid
    _checkStatus(outstream.device());

    // Make sure we leave room at the end of the last entry
    outstream << endl;
    }

void SimplePrefsSerializer::_saveEntry(Prefs& prefs, const VarElement &entry, const GroupElement &group)
    {
    // 1) create the string to save
    QString groupName = group.getFullGroupName();

    QString keyValueLine = "";
    QString descLine = "";
    QString entryLine = "";
```

```
if (!groupName.isEmpty())
        {
        entryLine += groupName;
        entryLine += ".";
        }

try {
// lets get the property now that we have the group name and all
const Property& prop = prefs.getProperty(entry.getName(), groupName);

  entryLine += entry.getName();
entryLine += "<";
keyValueLine = entryLine + prop.getConverter().typeName();
descLine = entryLine + "Description";
keyValueLine += ">=";
descLine += ">=";

// 2) create the appropriate key/value line
keyValueLine += prop.getConverter().toAscii( prop );
keyValueLine = keyValueLine.replace( QRegExp("\\n"), "~%~");

// 3) Create the description line
const QString desc = prefs.getDesc(entry.getName(), groupName);
if (desc != QString::null)
        {
        descLine += desc;
        descLine = descLine.replace( QRegExp("\\n"), "~%~");
        }
else
        descLine = QString::null;

// 4) save
fKeyValueStream << '\n' << keyValueLine ;

if (descLine != QString::null)
        {
        //qDebug("Writing desc");
        fDescStream << '\n' << descLine;
        }
}

catch (Exception e)
        {
        qDebug("SimplePrefsSerializer::_saveEntry: %s", e.toString().ascii());
        }
}
```

```
void SimplePrefsSerializer::_recursiveSave(Prefs& prefs, GroupElement *node)
  {
  // save all entries in this node
  //   qDebug("saving Entries for group '%s'", node->getFullGroupName().ascii());
  for (QListIterator<VarElement> entryIter = node->getVarEntryIterator();entryIter.current(); ++entryIter)
        _saveEntry(prefs, *entryIter.current(), *node);

  // save all children
  for (QListIterator<GroupElement> childrenIter = node->getSubgroupIterator();childrenIter.current();
++childrenIter)
        _recursiveSave(prefs, childrenIter.current());
  }

bool SimplePrefsSerializer::_cutOutCommentArea(QString &line, bool continuation)
  {
  int endCommentPos;
  int startCommentPos;

  if (continuation)
        {
        // Keep going until we find a close comment
        if ((endCommentPos = line.find("*/")) != -1)
          {
          line = line.right( line.length() - endCommentPos - 2); // subtract 2 for the size fo the */
          return true;
          }
        else
          {
          line = "";
          return false;
          }
        }

  else if ((startCommentPos = line.find("/*")) != -1) // Found C style comment
        {
        if ((endCommentPos = line.find("*/", startCommentPos)) != -1)
          {
          QString tempLine;
          tempLine = line.left(startCommentPos);
          tempLine += line.right(line.length() - endCommentPos-2);// subtract 2 for the size fo the */
          line = tempLine;
          return true;
          }
        else
          {
          line = line.left(startCommentPos);
          return false;
          }
```

```
        }

    else if ((startCommentPos = line.find("//")) != -1) // Found C++ style comment
        {
        line = line.left(startCommentPos);
        return true;
        }

    return true;
    }

 void SimplePrefsSerializer::load(Prefs& prefs, QIODevice &readableDevice)
    {
    bool continuation = false;

    if (!readableDevice.isReadable())
        throw IOException("Input/Output device is not readable");

    QTextStream instream( &readableDevice );

    QString curLine;
    int lineNum=0;
    while (!instream.atEnd())
        {
        _checkStatus(instream.device());
        curLine = instream.readLine();
        lineNum++;

        continuation = !_cutOutCommentArea(curLine, continuation);

        if (!curLine.isEmpty())
          _extractLine(prefs, curLine, lineNum);
        }
    }

/** parses the line and stores the information in the prefs class
    upon error, throws PrefsInvalidFormatException or UnrecognizedTypeException*/
void SimplePrefsSerializer::_extractLine(Prefs& prefs, QString line, int lineNum)
    {
    StringTokenizer tokenizer(line, ".<>{},=", true);
    TokenIterator tokenIter(tokenizer, true);

    //qDebug("SimplePrefsSerializer::_extractLine: Tokens in line:");
    //for (int i = 0; i < tokenIter.numTokens(); i++)
    //qDebug("\tToken [%d]: %s", i, tokenIter[i].ascii());

    try {
    tokenIter.toFirst();  // will throw exception if no tokens
```

```
    QString group  = _extractGroup(tokenIter, line, lineNum);
    QString name  = _extractName(tokenIter, line, lineNum);
    QString type =      _extractType(tokenIter, line, lineNum);

    // Check if it is a description
    if (type == "Description")
            {
            QString desc = _extractDesc(tokenIter, line, lineNum);
            _setPrefs(prefs, group, name, desc);
            }

    else // It is a key/value pair
            {
            Property *value = _extractValue(tokenIter, ConverterFactory::instance().getConverter( type ), line,
lineNum);
            _setPrefs(prefs, group, name, value);
            }
    }
    catch (TokenIteratorInvalidRangeException invalidRangeException)
            {
            throw PrefsInvalidFormatException(line, lineNum);
            }
    }

/**@param tokenIter precondition: tokenIter at the group.varName pos; postcondition: valid iter at var name token
   extracts the group from the tokens, checking for correct format;
   upon error thros PrefsInvalidFormatException
   @return the group string */
QString SimplePrefsSerializer::_extractGroup(TokenIterator &tokenIter, const QString &line, const int lineNum)

    {
    QString fullGroup="";
    QString aGroup;

    // iterate through the tokens until find the <, which signifies the type bracket
    while (tokenIter[tokenIter.getCurTokenNum()+1] != "<")
            {
            if (!fullGroup.isEmpty())
               fullGroup += ".";    // assign a dot seperator if not the first group

            aGroup = *tokenIter;  // assign the group
            fullGroup += aGroup;

            ++tokenIter;        // move to the next delimiter

            // if the next token isn't a valid delimiter or this isn't the correct delimiter, throw format exception
            if (*tokenIter != ".")
```

```
            throw PrefsInvalidFormatException(line, lineNum);

        ++tokenIter;    // move past the group seperator (. delimiter)
        }

    return fullGroup;
    }

/**@param tokenIter precondition: valid token at the name; postcondition: valid token at "<" delimtiter
  extracts the name from the tokens
  @return the name */
QString SimplePrefsSerializer::_extractName(TokenIterator &tokenIter, const QString &line, const int lineNum)
  {
  QString name = *tokenIter;

  ++tokenIter;
  if (*tokenIter != "<")
        throw PrefsInvalidFormatException(line, lineNum);
  return name;
  }

/**@param tokenIter precondition: iter at delimiter "<"; postcondition: valid iter one token past ">", the "=" operator
  extracts the type from the tokens, checking for correct format;
  upon error throws InvalidFormatException
  @return the property type */
QString SimplePrefsSerializer::_extractType(TokenIterator &tokenIter, const QString &line, const int lineNum)
  {
  // move passed the <
  ++tokenIter;

  // extract the type name
  QString propertyName = *tokenIter;
  propertyName = propertyName.stripWhiteSpace();

  // move to the > delim
  ++tokenIter;
  if (*tokenIter != ">")
        throw PrefsInvalidFormatException(line, lineNum);

  // move passed the > delim
  ++tokenIter;

  // the token; should be "=" delimiter or possibly some white space
  QString equalDelim = *tokenIter;
  equalDelim = equalDelim.stripWhiteSpace();
  if (equalDelim.isEmpty() || equalDelim.isNull())
        {
        tokenIter++;
```

```
        equalDelim = *tokenIter;
        }
    //qDebug("finished extracting type, getting equal delim '%s'", equalDelim.ascii());

    if (equalDelim != "=")
        throw PrefsInvalidFormatException(line, lineNum);

    return propertyName;
    }

/**@param tokenIter precondition: valid iter at token delim "="; postcondition: possibly invalid iter after value
   extracts the value from the tokens, checking for correct format;
   upon error throws PrefsInvalidFormatException
   @return the property type */
Property* SimplePrefsSerializer::_extractValue(TokenIterator &tokenIter, Converter& converter, const QString
&line, const int lineNum)
    {
    // Extract description gets all the tokens after the '=' and assembles them
    // into one long string. That is exactly what we would need to do here, so we
    // will just call that method
    return converter.fromAscii( _extractDesc(tokenIter, line, lineNum) );
    }

QString SimplePrefsSerializer::_extractDesc(TokenIterator &tokenIter, const QString &, const int)
    {
      // move passed the equals operator
    tokenIter++;
    QString valueString = "";

     //debug("Creating string:");
    int i = tokenIter.getCurTokenNum();
    while (i < tokenIter.numTokens())
        {
        //qDebug("\tAdding: %s", (*tokenIter).ascii());
        valueString += tokenIter[i];
        i++;
        }

    return valueString.replace( QRegExp("~%~"), "\n");;
    }

/** all parameters must be in correct format and valid
    calls the appropriate set method in the fPrefs */
void SimplePrefsSerializer::_setPrefs(Prefs& prefs, const QString &group, const QString &varName, Property
*value)
    {
    //qDebug("SimplePrefsSerializer::_setPrefs: Name: %s, Group: %s", varName.ascii(), group.ascii());
    prefs.setProperty(varName, value, group);
```

```
}

void SimplePrefsSerializer::_setPrefs(Prefs& prefs, const QString &group, const QString &varName, const QString
&desc)
  {
  prefs.setDesc(varName, desc, group);
  }
```

Appendix A.10 - PrefsEditor.h

```
#ifndef __PREFS_EDITOR_H
#define __PREFS_EDITOR_H
#include <qwidget.h>


/*****************************************************************************
 *****                                          *****
 *****          Classification: UNCLASSIFIED          *****
 *****           Classified By:                  *****
 *****           Declassify On:                   *****
 *****                                          *****
 *****************************************************************************
 *
 * $Source: /enews/proj/spg/cvsroot/CPL/GUI/Widgets/PrefsEditor.h,v $
 * $Author: mpilone $
 * $Date: 2000/02/01 19:59:13 $
 * $Id: PrefsEditor.h,v 1.1.1.1 2000/02/01 19:59:13 mpilone Exp $
 *
 * Developed by: Naval Research Laboratory, Tactical Electronic Warfare Div.
 *          Effectiveness of Navy Electronic Warfare Systems, Code 5707
 *          4555 Overlook Ave.
 *          Washington, D.C. 20375-5339
 *
 * For more information please call 202-767-2897 or send email to
 * enews@enews.nrl.navy.mil
 *
 * The U.S. Government retains all rights to use, duplicate, distribute,
 * disclose, or release this software.
 */

class Prefs;
class Property;
class ConverterWidget;
class GroupElement;
class VarElement;
class QListView;
class QListViewItem;
class QSplitter;
```

```
class QLabel;
class QScrollView;
class QVBoxLayout;
//class QVBox;
class CPL_EXPORT PrefsEditor : public QWidget
  {
   Q_OBJECT

  public :
   PrefsEditor(QWidget *parent, const char *name);
   virtual ~PrefsEditor();

   void setPrefs(Prefs &prefs);
   Prefs &getPrefs() const { return *fPrefs; }

   const QListView &getGroupTree() { return *fGroupTree; }
   const QListView &getVarList() { return *fVarList; }

   bool isPrefsModified() const { return fPrefsModified; }
   void setPrefsModified(bool v) { fPrefsModified = v; emit prefsModified(fPrefsModified); }
   void storeCurrentPropertyFromGUI();

   signals:
   void prefsModified(bool v);

protected slots:
   void slotUpdateVarDisplay(QListViewItem *groupTreeSelected);
   void slotUpdateValueEditorWidget(QListViewItem *varItemSelected);
   void slotPropertyModified();
   /**@param selectVar may be null
        designed to connect with PrefsFindDialog goButtonClicked, but could work with others */
   void slotUpdateDisplay(GroupElement *selectGroup, VarElement *selectVar);

  protected:
   static QStringList _getGroupStringList(const QListViewItem *groupTreeSelected);
   const Property &_getValueEditorProperty(QListViewItem *varItemSelected) const;
   void _setValueEditorWidget(const Property &p);
   void _showNothingSelected();
   void _selectVarItem(const QString &text);

  private :
   void _recursiveBuildGroupTree(GroupElement *node, QListViewItem *parent);
   QListViewItem *_openGroupTree(const QList<GroupElement> &branch);

   void _initPanes();
   void _initGroupTreePane();
   void _initVarListPane();
   void _initDescrValuePane();
```

```
  void _initConnections();
  void _setValueLabel( QListViewItem *varItem);

  /** coordinates switching widget editors and setting the property values */
  class EditWidgetCoordinator
        {
        public:
         /** store orig property, get new item widget, set as display */
         EditWidgetCoordinator(PrefsEditor *editor, QListViewItem *varItem, QListViewItem *groupItem);
         /** remove from display cur widget and store the new property if changed */
         ~EditWidgetCoordinator();

         /** updates the editor->fDescr text with the prefs desc */
         void showDesc();

         /** gets the value from the gui and stores in the prefs */
         void storePropertyFromGui();

         /** get the property from the prefs with the class group.var name */
         const Property &getProperty();

         ConverterWidget *getWidget() { return fWidget; }
         QListViewItem *getVarItem() { return fVarItem; }
         QListViewItem *getGroupItem() { return fGroupItem; }
         bool isPropertyModified() { return fPropertyModified; }
         void setModified(bool v) { fPropertyModified = v; }

        private :
         PrefsEditor *fEditor;

         QVBoxLayout *fLayout;
         ConverterWidget *fWidget;
         QListViewItem *fVarItem;
         QListViewItem *fGroupItem;
         bool fPropertyModified;
        };
friend class EditWidgetCoordinator;

 EditWidgetCoordinator *fEditWidgetCoordinator;

 QListView *fGroupTree;
 GroupElement *fGroupRoot;
 QListView *fVarList;

 QSplitter *fVarValueSplitter;
 QSplitter *fTreeVarSplitter;

 QLabel *fDescrLabel;
```

```
    QLabel *fDescr;
    QLabel *fValueLabel;

    //QScrollView *fDescrValueScroll;
    QWidget *fValueContainerWidget;
    QWidget *fRightPanelWidget; // contains the description and value

    Prefs *fPrefs;

    bool fPrefsModified;
  };

#endif
```

Appendix A.11 - PrefsEditor.cpp

```
#include "CPL/GUI/Widgets/PrefsEditor.h"

#include <qsplitter.h>
#include <qlabel.h>
#include <qlistview.h>
#include <qvbox.h>
#include <qscrollview.h>
#include <qlayout.h>

#include "CPL/Convert/Converter.h"
#include "CPL/Convert/Property.h"
#include "CPL/Prefs/Prefs.h"
#include "CPL/Prefs/GroupElement.h"
#include "CPL/Prefs/VarElement.h"
#include "CPL/Prefs/Prefs.h"

PrefsEditor::PrefsEditor(QWidget *parent, const char *name) : QWidget(parent, name), fPrefsModified(false),
fPrefs(0L)
  {
  _initPanes();
  _initGroupTreePane();
  _initVarListPane();
  _initDescrValuePane();

  fEditWidgetCoordinator = 0L;
  _initConnections();
  _showNothingSelected();
  }

PrefsEditor::~PrefsEditor()
  {
  delete fEditWidgetCoordinator;
```

```
    }

 void PrefsEditor::_initConnections()
    {
    QObject::connect(fGroupTree, SIGNAL( selectionChanged( QListViewItem *) ), this, SLOT(
 slotUpdateVarDisplay(QListViewItem *) ));
    QObject::connect(fVarList, SIGNAL( selectionChanged( QListViewItem *) ), this, SLOT(
 slotUpdateValueEditorWidget(QListViewItem *)));
    }

 void PrefsEditor::_initPanes()
    {
    fVarValueSplitter = new QSplitter(this, "fVarValueSplitter");
    fVarValueSplitter->setOrientation(QSplitter::Horizontal);

    fTreeVarSplitter = new QSplitter(fVarValueSplitter, "fTreeVarSplitter");
    fTreeVarSplitter->setOrientation(QSplitter::Vertical);

    fRightPanelWidget = new QWidget(fVarValueSplitter, "fRightPanelWidget");

    QVBoxLayout *layout = new QVBoxLayout( this );
    layout->addWidget(fVarValueSplitter);
    }

 void PrefsEditor::_initGroupTreePane()
    {
    fGroupTree = new QListView(fTreeVarSplitter, "fGroupTree");
    fGroupTree->addColumn("Group");
    fGroupTree->addColumn("# Items");
    fGroupTree->setMinimumSize( 110, 100);
    }

 void PrefsEditor::_initVarListPane()
    {
    fVarList = new QListView(fTreeVarSplitter, "fVarList");
    fVarList->addColumn("Name");
    fVarList->setMinimumSize( 110, 100);
    }

 void PrefsEditor::_initDescrValuePane()
    {
    QVBoxLayout *layout = new QVBoxLayout(fRightPanelWidget);

    fDescrLabel = new QLabel("Description", fRightPanelWidget, "fDescrLabel");
    fDescrLabel->setFont(QFont("Times", 14, QFont::Bold));
    fDescrLabel->setAlignment( QObject::AlignCenter );
    fDescrLabel->setMaximumHeight( fDescrLabel->sizeHint().height() );
    layout->addWidget(fDescrLabel);
```

```
    fDescr = new QLabel("", fRightPanelWidget, "fDescr");
    fDescr->setFont( QFont("Times", 12, QFont::Normal) );
    fDescr->setAlignment( QObject::AlignCenter );
    fDescr->setAutoResize( true );
    fDescr->setMinimumSize( 100, 150);
    fDescr->setMaximumHeight( 150 );
    layout->addWidget(fDescr);

    QFrame *lineSeperator = new QFrame( fRightPanelWidget, "descr/value line seperator");
    lineSeperator->setFrameStyle ( QFrame::HLine | QFrame::Sunken);
    lineSeperator->setLineWidth( 2 );
    lineSeperator->setMidLineWidth( 1 );
    lineSeperator->setFixedHeight( 10 );
    layout->addWidget(lineSeperator);

    fValueLabel = new QLabel("Value", fRightPanelWidget, "fValueLabel");
    _setValueLabel(0L);
    fValueLabel->setFont( QFont("times", 14, QFont::Bold) );
    fValueLabel->setAlignment( QObject::AlignCenter );
    fValueLabel->setMaximumHeight( fValueLabel->sizeHint().height() );
    layout->addWidget(fValueLabel);

    fValueContainerWidget = new QWidget( fRightPanelWidget, "fValueContainerWidget");
    layout->addWidget(fValueContainerWidget);
    }

void PrefsEditor::_setValueLabel(QListViewItem *viewItem)
    {
    QString label="";
    if (viewItem)
            {
            label = viewItem->text(0);
            label += " Value";
            }
    else
            label = "No Item Selected";
    fValueLabel->setText( label );
    }

QStringList PrefsEditor::_getGroupStringList(const QListViewItem *groupTreeSelected)
    {
    QStringList groupName;
    while (groupTreeSelected->parent())
            {
            groupName.insert( groupName.begin(), groupTreeSelected->text(0) );
            groupTreeSelected = groupTreeSelected->parent();
            }
```

```
    return groupName;
    }

 void PrefsEditor::slotUpdateDisplay(GroupElement *selectGroup, VarElement *selectVar)
    {
    QString fullGroupName = selectGroup->getFullGroupName().ascii();
    //if (selectVar)
    //qDebug("PrefsEditor::slotUpdateDisplay(%s,%s)", fullGroupName.ascii(), selectVar->getName().ascii());
    //   else
    //qDebug("PrefsEditor::slotUpdateDisplay(%s, null )", fullGroupName.ascii());
    QList<GroupElement> branch;

    fGroupRoot->findChild(Prefs::parseGroupVarString(fullGroupName,false), &branch);
    //qDebug("\tbuilt branch from tree; branch has %d nodes", branch.count());
    QListViewItem *groupItem = _openGroupTree(branch);
    if (groupItem)
            {
            fGroupTree->setSelected(groupItem, true);
            if (selectVar)
               _selectVarItem(selectVar->getName());
            }
    }

 void PrefsEditor::_selectVarItem(const QString &text)
    {
    bool itemSelected=false;
    for (QListViewItem *varItem=fVarList->firstChild();varItem && !itemSelected;varItem=varItem->nextSibling())
            if (varItem->text(0) == text)
               {
               fVarList->setSelected(varItem, true);
               itemSelected=true;
               }
    }

QListViewItem *PrefsEditor::_openGroupTree(const QList<GroupElement> &branch)
    {
    bool itemOpened=true;
    QListViewItem *childItemFound=0L;
    QListViewItem *firstInLevel=fGroupTree->firstChild();
    if (branch.count() == 0)
            return firstInLevel;
    firstInLevel=firstInLevel->firstChild();
    for (QListIterator<GroupElement> branchIter(branch);branchIter.current() && itemOpened;++branchIter)
            {
            itemOpened=false;
            //qDebug("\ttrying to open tree node for group = '%s'",branchIter.current()->getGroupName().ascii());
            for (QListViewItem *groupItem=firstInLevel;groupItem && !itemOpened;groupItem=groupItem-
>nextSibling())
```

```
            {
            //qDebug("\tcomparing item %s", groupItem->text(0).ascii());
            if (groupItem->text(0) == branchIter.current()->getGroupName())
                {
                //qDebug("\topening tree node = '%s'", groupItem->text(0).ascii());
                if (childItemFound)
                  childItemFound->setOpen(true);
                childItemFound=groupItem;
                firstInLevel=childItemFound->firstChild();
                itemOpened=true;
                }
            }
          }
  return childItemFound;
  }

void PrefsEditor::slotUpdateVarDisplay(QListViewItem *groupTreeSelected)
  {
  // delete the current list
  fVarList->clear();

  if (groupTreeSelected)
        {
        QStringList groupList = _getGroupStringList(groupTreeSelected);
        //qDebug("slotUpdateVarDisplay; building var list for group '%s'",
Prefs::createGroupVarString(groupList).ascii());
        // get the node
        GroupElement *groupNode = fGroupRoot->findChild(groupList);

        // build the var list from the node
        VarElement *aVarEntry;
        QListViewItem *listItem;
        QListIterator<VarElement> varIter = groupNode->getVarEntryIterator();
        for (varIter.toFirst();varIter.current();++varIter)
          {
          aVarEntry = varIter.current();
          //qDebug("\tadding var item '%s", aVarEntry->getName().ascii());
          listItem = new QListViewItem(fVarList, aVarEntry->getName(), "listItem");
          listItem->setOpen(true);
          }

        // select an item in the fVarList
        fVarList->setSelected( fVarList->firstChild(), true);
        }
  }

void PrefsEditor::slotUpdateValueEditorWidget(QListViewItem *varItemSelected)
  {
```

```
    //if (varItemSelected)
    //qDebug("slotUpdateValueEditorWidget for %s", varItemSelected->text(0).ascii());
    //   else
    //qDebug("slotUpdateValueEditorWidget for <empty item>");

    delete fEditWidgetCoordinator;
    fEditWidgetCoordinator = 0L;
    _setValueLabel( varItemSelected );
    if (varItemSelected)
            fEditWidgetCoordinator = new EditWidgetCoordinator(this, varItemSelected, fGroupTree->selectedItem());

    }

void PrefsEditor::setPrefs(Prefs &prefs)
    {
    if (fPrefs)
            {
            fVarList->clearSelection();
            fGroupTree->clearSelection();

            delete fGroupRoot;
            fGroupRoot = 0L;
            }

    fPrefs = &prefs;
    fPrefsModified = false;

    // reset the group tree
    //qDebug("setPrefs; creating group list");
    fGroupTree->clear();
    fGroupRoot = fPrefs->getGroupElementTree();

    _recursiveBuildGroupTree( fGroupRoot, 0L );

    fGroupTree->setOpen( fGroupTree->firstChild(), true);
    }

void PrefsEditor::_recursiveBuildGroupTree(GroupElement *node, QListViewItem *parent)
    {
    // add the node as an entry in the tree
    QListViewItem *listItem;
    if (parent)
            listItem = new QListViewItem(parent, node->getGroupName(), QString::number(node-
>getVarEntryIterator().count()));
    else
            listItem = new QListViewItem(fGroupTree, ".(root)", QString::number(fGroupRoot-
>getVarEntryIterator().count()));
```

```
  //qDebug("\tadded item '%s' (full=%s)", node->getGroupName().ascii(), node->getFullGroupName().ascii());

  // build the tree nodes for all children (ie sub groups)
  for (QListIterator<GroupElement> childrenIter = node->getSubgroupIterator();childrenIter.current();
++childrenIter)
        _recursiveBuildGroupTree(childrenIter.current(), listItem);
  }

void PrefsEditor::slotPropertyModified()
  {
  if (fEditWidgetCoordinator)
        fEditWidgetCoordinator->setModified(true);
  setPrefsModified(true);
  }

void PrefsEditor::_showNothingSelected()
  {
  fDescr->setText("No item selected");
  }

void PrefsEditor::storeCurrentPropertyFromGUI()
  {
  if (fEditWidgetCoordinator)
        fEditWidgetCoordinator->storePropertyFromGui();
  }

/** add the widget that corresponds to the selected group,var item to the edit widget area */
PrefsEditor::EditWidgetCoordinator::EditWidgetCoordinator(PrefsEditor *editor, QListViewItem *varItem,
QListViewItem *groupItem) :
    fEditor(editor), fVarItem(varItem), fGroupItem(groupItem), fPropertyModified(false)
  {
  fLayout = new QVBoxLayout(fEditor->fValueContainerWidget);
  fLayout->setMargin( 10 );

  const Property &p = getProperty();
  fWidget = p.getConverter().toGUI(p, fEditor->fValueContainerWidget, "EditWidgetCoordinator::fWidget");

  fLayout->addWidget(fWidget);

  fWidget->show();
  QObject::connect( fWidget, SIGNAL(modified()), fEditor, SLOT(slotPropertyModified()));

  showDesc();
  }

PrefsEditor::EditWidgetCoordinator::~EditWidgetCoordinator()
```

```
    {
    //qDebug("deleting EditWidgetCoordinator for %s", fVarItem->text(0).ascii());
    storePropertyFromGui();
    //qDebug("\tdisconnecting");
    QObject::disconnect( fWidget, SIGNAL(modified()), fEditor, SLOT(slotPropertyModified()) );
    delete fWidget;
    fWidget = 0L;
    delete fLayout;
    fLayout = 0L;
    fEditor->_showNothingSelected();
    }


void PrefsEditor::EditWidgetCoordinator::showDesc()
    {
    QString groupStr = Prefs::createGroupVarString( _getGroupStringList( fGroupItem ) );
    QString propertyName = fVarItem->text(0);
    QString descr = fEditor->getPrefs().getDesc( propertyName, groupStr );
    if (descr.isEmpty() || descr.isNull())
            fEditor->fDescr->setText("No description");
    else
            fEditor->fDescr->setText( descr );
    }

const Property &PrefsEditor::EditWidgetCoordinator::getProperty()
    {
    QString groupStr = Prefs::createGroupVarString( _getGroupStringList( fGroupItem ) );
    QString propertyName = fVarItem->text(0);
    return fEditor->getPrefs().getProperty( propertyName, groupStr);
    }

void PrefsEditor::EditWidgetCoordinator::storePropertyFromGui()
    {
    try {

    //qDebug("EditWidgetCoordinator::storePropertyFromGui()");
    if (isPropertyModified())
            {
            // gather all parameters to set the property from the gui
            QString propertyName = fVarItem->text(0);
            QString groupString = Prefs::createGroupVarString( _getGroupStringList( fGroupItem ));
            const Property &p = getProperty();
            //qDebug("\tgathering parameters; property name=%s\tgroup string=%s", propertyName.ascii(),
groupString.ascii());
            Property *newProperty = p.getConverter().fromGUI(*fWidget);

            //qDebug("\tsetting property %s.%s = %s", groupString.ascii(), propertyName.ascii(), newProperty-
>getConverter().toAscii(*newProperty).ascii());
```

```
            // set the property
            fEditor->getPrefs().setProperty(propertyName, newProperty, groupString);

            // emit signal that the prefs have been modified
            emit fEditor->prefsModified(true);

            // reset the property modified flag
            fPropertyModified = false;
            }

        }
        catch (Exception e)
            {
            qDebug(e.toString().ascii());
            }
        }
```