GINA++

The Generic Interactive Application for C++ and OSF/Motif

User Manual and Tutorial

Version 1.6

Andreas Bäcker, Andreas Genau and Markus Sohlenkamp

German National Research Center for Computer Science
Institute for Applied Information Technology (FIT)
Human-Computer Interaction Research Division (HCI)
P.O. Box 1316, D-5205 St. Augustin 1
email: `baecker@gmd.de`, `genau@gmd.de`, `sohlenkamp@gmd.de`

December 16, 1992

Contents

1 Introduction

GINA++ (**G**eneric **IN**teractive **A**pplication for C++) is an object-oriented *application framework* that facilitates the development of applications having a graphical user interface. It is written in C++ and uses OSF/Motif and the X Window System for the user interface parts. GINA++ comes as a set of C++ classes plus a set of demo applications.

The classes of GINA++ constitute an executable program which possesses the basic functionality and the user interface components identical to all interactive graphical applications, but lacks any application-specific behavior. New applications are constructed according to the object-oriented software development paradigm: The predefined functionality and behavior is inherited by deriving subclasses from the existing GINA++ classes. Application specific behavior is implemented by adding new classes and member functions. Differences to the standard application are coded by refining virtual member functions inherited from GINA++.

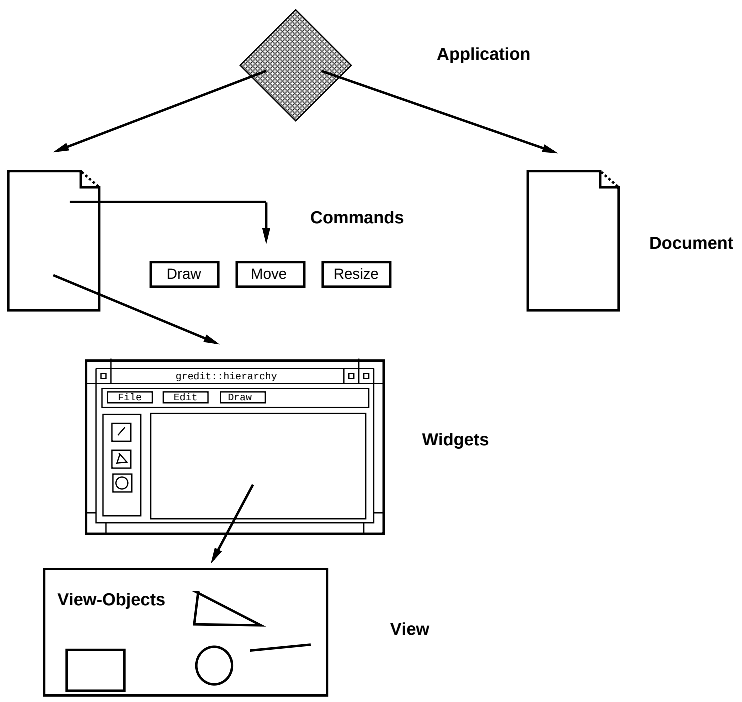

Figure 1.1: Object hierarchy in a GINA++ application

The object structure of a typical GINA++ application is shown in figure 1.1: Each application has exactly one object of the class `GnApplication` and one object of a user-defined subclass of `GnDocument` for each document currently being processed. A list of all document objects is stored in the application object. In general, there is at least one instance of a user-defined subclass of `GnView` for each document. The document objects store a list of all views. The graphical user interface is composed through a set of widgets, which are instances of subclasses of `GnWidget`. A view may contain a set of graphical objects, which are modelled by user-defined subclasses of `GnViewObject` and `GnDirectManipulationObject`. The document also stores the list of previously executed commands, which are modelled through subclasses of `GnCommand` and `GnMouseDownCommand`. The inheritance relationship of the classes described so far is shown in figure 1.2.

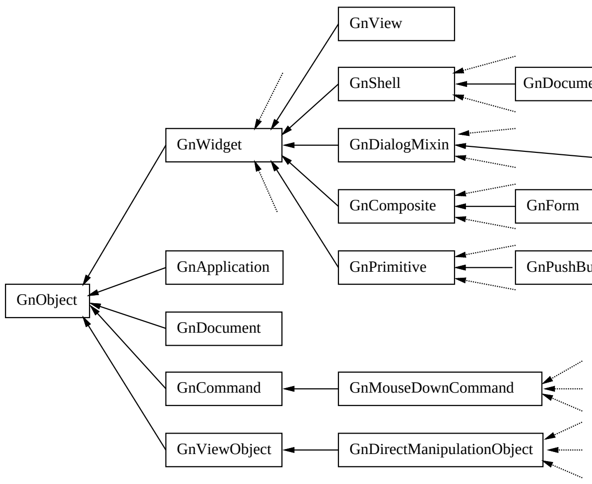

Figure 1.2: Part of the class hierarchy of GINA++

The generic interactive application maintains an arbitrary number of *documents*, each displayed in a specific *document shell*. A document shell forms the main window of an application and is composed of the *widgets* of OSF/Motif. In general, it contains a *menu bar* and a set of *widgets* and *views* which display the contents of the document and provide means for *user interaction* according to the principle of *direct manipulation*.

The menu bar already has a “File” and “Edit” topic, and – if the application was compiled with debugging options – an additional “Debug” topic. The “File” topic contains entries for opening, closing, saving and creating documents as well as an entry for generic printing of documents. The “Edit” topic contains entries for unlimited undoing and redoing of user commands, for animated replay of the command history, and for activating the *history scroller*. It also contains entries for cut, copy, and paste operations. The “Debug” topic, if present, contains an entry for popping up a dialog box that displays the hierarchy of widgets.

In general, the Motif widgets are not suitable to represent the contents of a document and do not cover all required interaction styles and techniques. This gap is bridged by views, which provide both a *procedural* and an *object oriented* interface to graphic output as well as an interface for direct-manipulation interaction techniques. The procedural interface consists of a set of basic drawing functions and maintains default graphic contexts as well as default fonts. The object-oriented interface is built on top of the procedural one and provides support for graphical view-objects such as rectangles, circles and bitmap icons. View-objects maintain their own size and position, can be moved and resized by the application code and are redrawn by GINA++ whenever necessary. *Direct-manipulation-*

objects, a subclass of view-objects, can be selected, moved around in a view, and resized by the user. A subframework inside GINA++ supports the implementation of new types of view-objects and direct-manipulation objects.

All kinds of user actions and interactions are modelled through *command classes*. A command class implements the intended effects of an operation, and optionally implements the inverse operation, building the base for unlimited *undoing* and *redoing*. A *command object* is created for each user action and is submitted to and maintained by GINA++. The data members of command objects hold the state information necessary to undo and redo the associated operation. GINA++ stores a list of all submitted command objects on a per-document basis. The "Undo" and "Redo" menu entries in the "Edit" menu topic are used to undo or redo single commands. The *history scroller* dialog provides push buttons for single undo and redo and a scrollbar for undoing and redoing multiple commands at a time. The selection of the menu entry "Replay History" starts a replay of the command history. User actions with graphical feedback (inside views) are modelled through *mouse-down-command* classes. A mouse-down-command object can be created when a mouse button is pressed inside a view, and displays a *graphical feedback* as long as the mouse is dragged around. When the mouse button is released, the mouse-down-command object is submitted to GINA++ and is treated as all normal command objects. Again, a subframework inside GINA++ supports the implemenation of new command and mouse-down-command classes.

The classes of GINA++ are arranged into three layers: The basic layer consists of a *meta-object-protocol*, basic *data structures* and *iterators*, and wrapper classes for strings and all primitive data types of C++ (`int`, `float`, etc.). The next layer provides an encapsulation of OSF/Motif, and the third layer adds the application framework classes like documents, views and commands.

The meta-object-protocol provides information at run-time about all classes that make up an application. These informations are contained in the program's static data area. Information is stored about the inheritance graph, the names of classes and the size of instances of a class. The meta-object-protocol provides the basis for secure type casts, for querying the dynamic type of instances as well as for generic operations like loading and saving documents.

Data structures include linked lists and an iterator class for each list type. All data types are realized as parameterized type, i.e. the type of the elements that can be stored in data structures is a parameter of the class definition. The instantiation of type parameters (i.e. the definition of data structures that constrain their elements to be of a specific type) is supported by macros in a fashion similar to the proposed *template* facility of C++.

The wrapper classes for strings and primitive data types allow to treat instances of these types as true class objects. This minimizes the need for type casts and enables the derivation of subclasses of primitive types.

A set of base classes is provided to model dependencies between objects and to allow the propagation of changes to related objects.

The widget classes of OSF/Motif have been encapsulated into C++ classes [Bae91a], with a corresponding C++ class for each widget class in the Motif and X toolkits. The main purpose of this encapsulation is to provide a software framework for the design of new complex and reusable widget classes completely in C++, without having to deal with the object-oriented mechanisms of the X Toolkit Intrinsics, which are merely based on programming conventions. The encapsulation also enforces strict type-checking for widget resources and a functional interface for modifying and accessing resources. The callback mechanism of the X toolkit has been adapted in a way that all types of class member functions, especially non-static functions, can be attached as callback functions. Whenever reasonable, callbacks have been hidden and mapped internally on calls to virtual functions, which can be redefined in subclasses when required.

A special set of classes deals with the Motif menu system, allowing the easy construction of different types of menus, using both the standard menus entries supported by Motif as well as more sophisticated ones provided by GINA++.

The GINA++ classes have been developed systematically according to the principle of *design by contract*[Mey88], which was first realized in the *Eiffel* programming language [Mey92]. Design by contract views the construction of a software system as the fulfillment of contracts between clients and suppliers, each having rights and obligations. Contracts are expressed through *preconditions* and *postconditions* in functions and through *class invariants*. GINA++ adapts design by contract to C++ and provides means to express contracts betwen client and supplier classes.

The rest of this document is arranged as follows: Chapter 2 gives an introduction to the principles of design by contract and how to apply them. Chapter 3 describes the facilities of the meta-object protocol. Chapter 4 describes the generic data structures and shows how to use them effectively. Change Propagation is the topic of chapter 5. In chapter 6 the encapsulation of OSF/Motif is explained. A more detailed description of the Motif encapsulation and the interaction types build on top of it can be found in [Bae91b] and [Bae91a]. The menu system is described in Chapter 7. Chapter 8 contains a description of the view and the view objects. In chapter 9 the undo facility and the command object mechanism is explained. A comprehensive tutorial on application development with GINA++ can be found in chapter 10, the source code of the example application being contained in appendix A.

It is not necessary to read the manual in this order to understand the basic concepts of GINA++. Instead, you can start with any chapter you are interested in. For a quick start and an overview of the most important features, read the tutorial and the example source code — this should provide you with enough information to write your own simple applications with GINA++.

To get a better description of the most important classes within GINA++, look at chapters 6, 8 and 9. The other chapters provide you with information necessary to use some advanced features of GINA++.

2 Design by Contract

2.1 Introduction

The principle of *Design by Contract* has been first introduced within the *Eiffel* language (see [Mey92] and [Mey88]). Its purpose is to enhance software quality in terms of reliability and correctness by providing a systematic approach for designing classes. Besides that, it aids in program debugging and forms a basis for documentation and program verification.

Design by contract views the construction of a software system as the fulfillment of contracts between *clients* and *suppliers*, each having rights and obligations. In an object-oriented software system, each class may act as a supplier offering services. Client classes take these services either by *inheriting* from a supplier class or by *using* (i.e., creating instances and calling member functions) a supplier classes's operations.

Contracts between clients and suppliers are expressed through *preconditions* and *postconditions* in routines and functions, and through *class invariants* . They are realized by program assertions, written as boolean expressions.

- The preconditions state the conditions under which a client can apply a supplier's function. Any clients must guarantee the preconditions when calling a supplier's function. This implies that suppliers must provide means for clients to test the preconditions of a function. A function call that fulfills the preconditions is called a *correct call*.
- The postconditions state the properties a function guarantees at the completion of a correct call. If a client fulfills his part of the contract by performing a correct call, then the supplier in turn fulfills his part of the contract by ensuring the postconditions.
- The class invariant specifies properties which any instance of a class must satisfy. Each constructor function must create an object that satisfies the class invariant, and any non-private function or routine must transform the object into a state that again satisfies the class invariant.

A class is then said to be consistent if its implementation satisfies the correctness requirements expressed by the preconditions on the routines and functions of the class, the postconditions, and the class invariant (see [Mey92], page 127).

The *specification* of a function then consists of its signature and its pre- and postconditions. The signature is determined by the functions name and the types of its arguments and its return value. This notion requires a special treatment of pure virtual functions, which is discussed in section 2.10.

2.2 Design by Contract with C++

The GINA++ class library provides a set of preprocessor macros for defining assertions and for checking class invariants.
These macros are contained in the file `include/GnBasic/GnContracts.h`.

Because checking of assertions and class invariants may cause a significant overhead, and because it is usually only desired during the software testing and verification phase, it may be enabled and disabled by the preprocessor option `-DCONTRACTS`. If this option is *not* specified, then all macros expand into nothing and all checking of assertions and class invariants is disabled and no extra code is produced.

When a class is compiled with the `-DCONTRACTS` option, then all assertions and class invariants are checked. Violating an assertion or a class invariant causes an error message to be printed on `cerr`, and a core dump is produced.

Compiling code with the `-DCONTRACTS` option makes it in general unnecessary to specify the debug option `-g` of the compiler. As a great benefit, this results in a significantly shorter edit-compile-test cycle, because the linking time required is much shorter compared to those required with `-g`. Additionally, core dumps can be loaded into the debugger much quicker.

Experience has shown that most errors can be located by examining the error message that is produced when an assertion is violated, plus the backtrace of the stack of the program. So we recommend the following development style: Compile your programs only with `-DCONTRACTS`. Find errors by examining error messages and backtraces. If an error can not be detected this way, compile your program again with both `-DCONTRACTS` and `-g` specified, and again try to find the error. GINA++ contains three versions of its class library:

- `libGINA.a` is an optimized version. It should be used for the product version of your software.
- `libGINA_d.a` is compiled with `-DCONTRACTS`. It should be used during normal program development and testing.
- `libGINA_dg.a` is compiled with both `-DCONTRACTS` and `-g`. It should be used when searching serious bugs.

Every version of the GINA++ library a shared library can be generated by setting the appropriate makefile options on installation.

2.3 An Example

In the following discussion, the design of a class according to the principle of *Design by Contract* is shown at the example of the GINA++ class `GnPresentable` and two of its subclasses, `GnBitmapFilePresentable` and `GnBitmapBitsPresentable`.

```
class GnPresentable : public GnObject {
  public:
    GnPresentable();
  public:
```

```
    virtual int IsStringPresentable() { return True; }
    virtual int IsBitmapPresentable() { return False; }
    virtual int HasBitmapFile()       { return False; }
    virtual int HasBitmapBits()       { return False; }
  public:
    virtual char *PrintOn(char *buffer, unsigned int length);
    virtual char *BitmapFileName();
    virtual char *BitmapBits(unsigned int &width, unsigned int &height);
};
```

The class `GnPresentable` is an (abstract) base class that covers aspects of presenting objects. An object may be presented through

- a text string, for example person names in a scrollable selection list. The object's textual representation is obtained by calling `PrintOn()`, and by passing in a character buffer and its length for printing it on.
- a bitmap icon, for example fill patterns in the palette of a graphical editor. The data describing the bitmap icon may be accessed by calling either `BitmapFileName()` or `BitmapBits()`, depending on whether the data is stored in a file or in main memory.
- both a text and an icon, for example file and directory names in a graphical shell.

In the remainder of this chapter we elaborate a more reasonable class interface that conforms to the principles of *Design by Contract*.

The two subclasses `GnBitmapFilePresentable` and `GnBitmapBitsPresentable` provide means for obtaining the representation of objects that are presentable through bitmaps. The first class deals with bitmaps stored in files, the second one with bitmaps statically stored in main memory. Both classes refine virtual function which they inherited from their base class `GnPresentable`. Later in this chapter, we will use these functions to discuss rules for and constraints on defining assertions and class invariants.

2.4 Defining Assertions

In general, an assertion consists of a boolean expression and an (optional) informal comment describing its semantics. The comment must be a null-terminated character string (`char *`). Additional constructs are used to save values of variables, data members and expressions at function entry for later use in the postcondition to state the changes a function causes.

2.4.1 Preconditions and Postconditions

The two macros `REQUIRE` and `ENSURE` are used to state preconditions and postconditions, respectively. As a convenience, the preconditions are always the first statements in a function, and the postcondition is always the last statement, except when followed by a `return` statement. We now show how they can be used to define them for the function `PrintOn()`:

```
char *GnPresentable::
PrintOn(char *buffer, unsigned int length)
{
    REQUIRE( IsStringPresentable(), "Object is presentable by a text string");
    REQUIRE( buffer != 0,           "Buffer is not NULL");
    REQUIRE( length > 0,            "Buffer length is positive");

    ostrstream str(buffer, length, ios::out);

    str << (void *)this << '\0';
    buffer[length-1] = '\0';

    ENSURE( buffer[length-1] == '\0', "Text string is NULL-terminated");

    return buffer;
}
```

The precondition consists of three assertions: The first one states that the object really has a textual representation, the second one states that the buffer must not be a NULL pointer, the third one states that the buffer must have at least one byte.

The postcondition states that the text string is NULL-terminated, and any function which processes this string may savely rely on this. It is important to note that the third part of the precondition is essentiell for being able to guarantee the postcondition: If we would allow zero-length strings, then we would not be able to store a NULL-byte in all cases and therefore could not ensure the postcondition.

The postcondition does not state anything about the result being returned from `PrintOn()`, although it is obvious that the result is the buffer pointer passed in as an argument. If the postcondition should state something about the result of the function, it is often necessary to store it on a temporary variable, because otherwise it would not be accessible in the postcondition. As a convenience, this temporary variable should be named `Result`. The last two statements of `PrintOn()` would then look like:

```
    char *Result = buffer;

    ENSURE( Result[length-1] == '\0', "Text string is NULL-terminated" );
    ENSURE( Result == buffer, "The buffer being passed in is the result" );

    return Result;
```

On the first glance, this accurate postcondition might look superfluos, but there are actually three good reasons for doing so: First, the postcondition should state as much as possible about a functions effect, in order to provide as much information as possible to the programmer. Second, the pre- and postconditions might serve as the basis for class documentation, so the informal comments should document as much as possible of the function's behavior. The third and most important reason is that subclasses also "inherit" the pre- and postconditions of virtual functions and must conform to them. This topic will be discussed more precisely in section 2.6.

2.4.2 Describing State Changes

In the above example, the assertion

```
ENSURE( Result == buffer, "The buffer being passed in is the result" );
```

does *not* state that the function's return value is equal to the value that was passed in as an argument. Because the value of `buffer` could have been modified during function execution, it only guarantees that the value of `Result` is equal to the value of `buffer` by the time the `return` statement is executed.

If we want to be precise, we must save the original value of `buffer` on function entry, and compare this value against `Result` in the postcondition. GINA++ provides the macros `REMEMBER`, `OLD`, `REMEMBER_VALUE`, and `OLD_VALUE` for this purpose. They must only be used in preconditions and postconditions.

The macros `REMEMBER` and `REMEMBER_VALUE` are used to save values of variables and expressions, respectively. The arguments of `REMEMBER` are the type and name of the variable to be saved. The arguments of `REMEMBER_VALUE` are the type of the expression, the name of a variable to store the expression's value, and the expression itself.

The precondition of `PrintOn()` would then look like

```
REQUIRE( buffer != 0, "Buffer is not NULL");
REQUIRE( length > 0,  "Buffer length is positive");
REMEMBER( char *, buffer );
```

The macros `OLD` and `OLD_VALUE` are used to refer to values saved with `REMEMBER` and `REMEMBER_VALUE`. The argument to `OLD` is the name of the variable that has been saved, the argument to `OLD_VALUE` is the name of the variable where the expression has been saved on.

The postcondition of `PrintOn()` could then be extended to state that the return value is equal to the initial value of `buffer`:

```
ENSURE( Result == OLD(buffer), "The buffer being passed in is the result" );
```

2.4.3 Saving Values of Expressions

The macros `REMEMBER_VALUE` and `OLD_VALUE` could be used as follows: Assume a class `List`, which has a function `Length()`, which returns the current number of elements in the list, and a function `Remove()`, that removes an element from the list. If we want to ensure that the length of the list after removing an element is equal to the original length minus 1, we could do this as follows:

```
void List::
Remove(Element *element)
{
    REQUIRE( Contains(element), "Element is stored in the list");
    REMEMBER_VALUE(int, initial_length, Length());

    // Delete the element

    ENSURE( ! Contains(element), "Element is no more contained in the list");
    ENSURE( Length() == OLD_VALUE(initial_length) - 1,
            "One Element has been removed");
}
```

2.4.4 Implications in Preconditions and Postconditions

If we want to allow that `List::Remove()` could also be called with an element that is not contained in the list, we must relax the precondition and we need mechanisms to state postconditions conditionally. GINA++ provides the macros `REQUIRE_IF` and `ENSURE_IF` for this purpose. They both take an additional boolean expression as their first argument. They both require or ensure the second expression only if the first boolean expression holds. The function `Remove()` would then look like this:

```
void List::
Remove(Element *element)
{
    REMEMBER_VALUE(Boolean, contains, Contains(element) );
    REMEMBER_VALUE(int, initial_length, Length() );

    // Delete the element if it is contained in the list

    ENSURE( ! Contains(element), "Element is no more contained in the list");
    ENSURE_IF( OLD_VALUE(contains) == True,
               Length() + 1 == OLD_VALUE(initial_length),
               "One Element has been removed from the list");
}
```

2.4.5 Checking Conditions

The `CHECK` macro might be used by the class designer to check for unexpected behavior of his code. This macro is similar to `REQUIRE`: It checks a boolean expression and prints an error message if the expression evaluates to `False`. For example, the `CHECK` macro can be used in `PrintOn()` to test whether the string output stream `str` is in a good condition after printing on it:

```
    ostrstream str(buffer, length, ios::out);

    str << (void *)this << '\0';
    CHECK( str.good(), "stream is in a good condition" );
    buffer[length-1] = '\0';
```

The `CHECK` macro is intended for verifying the *expected* behavior of software. It is not part of a function's specification, i.e., it has nothing to do with its pre- and postconditions. It is also not meant as a mechanism for error or exception handling.

2.5 Verifying Preconditions

Any function, operator or data member used for expressing the precondition of a function must also be applicable to the clients of a class. This gives clients the chance to verify a function's preconditions before calling it, and to take some appropriate action if they are not fulfilled.

For a public member function, this means that everything used in its preconditions must be public as well. For protected member functions, all constructs used in its precondition must be either public or protected. Because this constraint can not be checked during class compilation, it is always the programmer's responsibility to ensure this.

In the example class `GnPresentable`, the function `IsStringPresentable()` is made public, so that all clients can effectively test whether an object has a textual representation before calling `PrintOn()`. The last two assertions only use predefined operators, which are available to all potential clients.

2.6 Assertions and Inheritance

Dynamic binding of routines at runtime imposes constraints on the pre- and postconditions of virtual functions that are redefined in subclasses. Since a client often does not know the exact type of an object, and therefore does not know which version of the virtual function is actually called, he tries to satisfy at least the precondition defined in the specification of the virtual function of the base class, and relies on that the function guarantees at least what is stated in this function's postcondition.

For virtual functions redefined in subclasses, this means that their preconditions must always be satisfied when the precondition in their base class in satisfied, and that their postcondition must at least guarantee what their base class guarantees. To be more precise: If *pre* and *post* are the pre- and postconditions of a virtual function f in a class C, and *pre*$'$ and *post*$'$ are the pre- and postconditions of a virtual function f' in a class C', where C' is a subclass of C and f' is a redefinition of f, then *pre* must logically imply *pre*$'$ and *post*$'$ must logically imply *post*.

To fulfill these requirements when developing with GINA++, we recommend the following procedure: If you want to redefine a virtual function in a subclass, then copy the base classes' implementation of this function, delete the code between the pre- and postcondition, and implement your own version of the function. In most cases, you will simply reuse the base classes assertions.

The availability of the base classes source code will in general be no problem if you are the author of the class, and it will also be no problem if the base class is a GINA++ class, because GINA++ is always delivered with source code.

2.7 Defining and Checking Class Invariants

A class invariant is a boolean function. If applied to an object, the class invariant return `True` if the object is in a valid state, and `False` if it is not.

Classes wishing to define a class invariant must inherit from the class `GnContracts` and should spezialize the inherited function `ClassInvariant()`. The macro `CHECK_OBJECT(object)` might then be used to check whether `object` is in a valid state. A call to the macro `CHECK_INVARIANT()` is equivalent to calling `CHECK_OBJECT(this)`, and is intended to be used by member functions to check whether the object they operate on is valid, and is most oftenly placed at the beginning of postconditions. `CHECK_INVARIANT()` and `CHECK_OBJECT` are special forms of the `CHECK` macro, where the boolean expression is always a class invariant.

If the class `GnPresentable` wishes to define a class invariant, its definition must be extended as follows:

```
class GnPresentable : public GnContracts {
    public:
      Boolean ClassInvariant();
    ....
};
```

The class invariant then states that any presentable object has at least a textual representation:

```
Boolean GnPresentable::
ClassInvariant()
{
    return IsStringPresentable();
}
```

In this case, the class invariant is very simple. This is because `GnPresentable` describes a very simple abstract concept. Concrete subclasses will define their own class invariants, which will usually be more complex.

2.8 Class Invariants and Inheritance

Since instances of subclasses may also be treated as instances of their base class, it is necessary that the class invariants of base classes do not only apply to their own instances, but also to instances of their subclasses. From a clients point of view, this guarantees that instances of subclasses have at least the same properties and the same behavior as instances of their base class.

The class invariants of subclasses are therefore obtained by concatenating the class invariants of their base class and their own one. Since C++ does not support class invariants, it is the programmer's responsibility to concatenate class invariants with the logical operator `&&`.

Consider the class `GnBitmapFilePresentable`, which realizes objects that are presentable by bitmap icons and take the bitmap data from a bitmap file. This class inherits from `GnPresentable`, specializes both `IsBitmapPresentable()` and `HasBitmapFile()` to return `True`, and redefines the function `BitmapFileName()`. It also redefines its class invariant, stating that the object is not only textually presentable, but also by a bitmap icon, and that the icon data is retrievable from a file:

```
Boolean GnBitmapFilePresentable::
ClassInvariant()
{
    return GnPresentable::ClassInvariant() &&
           IsBitmapPresentable() && HasBitmapFile();
}
```

Note that this class invariant does not say anything about the property that the bitmap data might be stored in (static) main memory. This makes it possible to derive a further subclass `GnBitmapFileAndBitsPresentable`, which has access to its bitmap data both form a file and in main memory. The class invariant of this class would then look like:

```
Boolean GnBitmapFileAndBitsPresentable::
ClassInvariant()
{
    return GnBitmapFilePresentable::ClassInvariant() && HasBitmapBits();
}
```

If we had specified that instances of class `GnBitmapFilePresentable` have no access to an in-core version of their bitmap data, i.e., by writing something like

```
    return GnPresentable::ClassInvariant() &&
           IsBitmapPresentable() && HasBitmapFile() && ! HasBitmapBits();
```

in the class invariant, then we would not have been able to create valid instances of the class `GnBitmapFileAndBitsPresentable`, because their class invariant contains a contradiction. So the general rule for writing class invariants is to state the things a class does, and not those it does not.

2.9 Violating Assertions and Class Invariants

Violating an assertion or a class invariant causes an error message being printed to `cerr`, and a core dump is produced. The error message tells what kind of assertion has been violated, the comment contained in the assertion, and the file name and the line number where the assertion is located. If a class invariant is violated, then additional information describing the invalid object is printed.

Violating a precondition means that an incorrect call was issued. The source of the error then lies in the client's code.

If a postcondition is violated, then the source of the error lies in the function itself, because otherwise the call to this function would have been an incorrect call, which then would have violated the precondition. In such cases, a bug report including a minimal demo program should be sent to the author of the class.

The violation of a class invariant reveals an error either in a member function or, in rare cases, in the class invariant itself. In both cases, it is the responsibility of the class designer to fix this bug. If all non-private member functions contain a call to `CHECK_INVARIANT` at the end of their postcondition, then the error must be in the last called member function, because the invariant checks guarantee that the object was in a valid state after the last but one call to a member function.

2.10 Dealing with Pure Virtual Functions

The purpose of pure virtual functions is to introduce a pattern for a concept, which is later implemented in various subclasses. Although the implementation might not be done in the abstract base class, it is often desirable to state the pre- and postconditions of pure virtual functions. Because it is not possible to do this without implementing the functions, pure virtual functions must be worked around: They must be implemented in a way that they only contain pre- and postconditions, with a `CHECK` macro of the form

```
    CHECK( False, "Pure virtual function called" );
```

between them. For example, instead of introducing the function `BitmapFileName()` as pure virtual in the class `GnPresentable`, we implement it as follows:

```
char *GnPresentable::
BitmapFileName()
{
    REQUIRE( IsBitmapPresentable(), "Object has a bitmap representation");
    REQUIRE( HasBitmapFile(), "Bitmap data is stored in a file");

    CHECK( False, "Pure virtual function called" );

    char *Result = 0;

    ENSURE( Result != 0, "Result is not NULL");
    ENSURE( IsValidFileName(Result), "File name is a valid UNIX path name");
    ENSURE( IsReadable(Result), "File is readable");

    return Result;
}
```

This documents what this function does and forces subclasses to conform to this behaviour according to the rules stated in section 2.6.

3 The Metaobject Protocol

3.1 Introduction

The *metaobject protocol* provides type information for C++ objects at run-time . Every object belonging to a subclass of `GnObject` contains a reference to a *metaclass object* . There exists exactly one metaclass object per class, and all instances of a class reference the same metaclass object. A metaclass object is an instance of class `GnClass`. It contains the name of the class, its base classes and subclasses, the size in bytes of its instances, the names of the files containing the class's source code, and a description of the data members of its instances. The description of a data member contains its name, its static type, its size in bytes, and its position inside the enclosing object.

The run-time type information may be used to query both the *static* and the *dynamic* type of objects, as well as to check if an object belongs to a class or to one of its subclasses (see section 3.3).

The type information also provides the base for secure type casts (see section 3.4), thus ensuring that only legal casts are performed.

3.2 Defining Metaclass Information for a Class

The definition of metaclass functionality for classes is supported by a set of preprocessor macros. These macros specify and implement the functions described in the following sections. Additionally, they declare the metaclass object associated with the class.

The metaclass functionality for a class with one non-virtual base class is defined as follows:

1. The class must be a direct or indirect subclass of `GnObject`.

2. The class definition must contain the macro `META_DEF_1`, with the first argument being the class name, and the second argument being the name of its base class. For example, the metaclass functionality for the class `GnLinkedList` must be defined as follows:

```
class GnLinkedList : public GnSequence {
    META_DEF_1(GnLinkedList,GnSequence);
    // other stuff
  private:
    GnLinkable *first;
    GnLinkable *last;
```

```
    int         length;
};
```

There must be no white space characters in the argument list of META_DEF_1.

3. The metaclass functionality must be implemented with the META_IMPL_1 macro. There must be exactly one matching META_IMPL_1 for each META_DEF_1. The call to META_IMPL_1 is usually put into the file containing the implementation of the class's member functions. Its first argument must be the class name, the second argument must be the name of the base class, and the third argument is an enumeration of the class members, separated by semicolons[1]. For example:

```
META_IMPL_1(GnLinkedList,GnSequence, PTR(first);PTR(last);SIMPLE(length) );
```

   There must be no white space characters before the second comma inside the argument list of META_DEF_1.

A definition of metaclass functionality without a matching implementation results in error messages about undefined symbols when linking the program.

3.2.1 Classes with Two Base Classes

Metaclass functionality for classes having two non-virtual base classes must be defined with the macros META_DEF_2 and META_IMPL_2, which both take the second base class as their third argument.

3.2.2 Classes with Virtual Base Classes

Metaclass functionality for classes having one virtual base class must be defined with the macros META_DEF_1_V and META_IMPL_1_V.

3.3 Type Queries

Each class *class* defines the following functions for querying and testing its type:

`static GnClass *ClassOf()`
: This function returns a pointer to the metaclass object of a class. Its purpose is to obtain a metaclass object of a named class, without requiring an existing object, by writing *class*`::ClassOf()`.

`virtual GnClass *Class()`
: This function returns a pointer to the metaclass object of an existing object. Its purpose is to obtain the dynamic type of an object, by writing `obj->Class()`.

[1] This feature is not implemented yet. Therefore, the third argument must be left out, with a comma after the second argument: `META_IMPL_1(GnLinkedList,GnSequence,)`.

The differences between the functions `ClassOf()` and `Class()` are as follows: `ClassOf()` returns the *static* type of objects, whereas `Class()` returns the *dynamic* type of objects . `ClassOf()` and `Class()` are defined for all class types. `ClassOf()` accesses a metaclass object of a named class, without requiring an object.

3.4 Secure Casts

The metaobject protocol provides for each class *class* the function

static *class* &TypeSafeDownCast(GnObject &obj)

static *class* *TypeSafeDownCast(GnObject *obj)

to perform secure type conversions. For convenience, this function is overloaded to handle both references and pointers. The function is intended to perform potentially unsafe explicit cast operations from a base class type to a subclass type. For example, the type cast

```
GnSequence *sequence = new GnLinkedList();
GnLinkedList *list = (GnLinkedList *)sequence;
```

would be replaced by the statement

```
GnLinkedList *list = GnLinkedList::TypeSafeDownCast(sequence);
```

which has the same effect as the C++ cast operator. Additionally, the function `TypeSafeDownCast()` checks whether the dynamic type of its argument is equal to its target class or a subclass of its target class, and prints an error message and aborts the program if it is not. For example:

```
GnSequence *sequence = new GnSequence();
GnLinkedList *list_1 = (GnLinkedList *)sequence;

GnLinkedList *list_2 = GnLinkedList::TypeSafeDownCast(sequence);
```

Both assignments are legal, although they assign an object of a base class to a variable of a subclass. The first type conversion does not detect the error at run-time, possibly being the source for serious errors that are in general hard to find. The second type cast detects the illegal conversion at run-time, producing an error message and a core dump.

3.5 Obtaining Information about a Class

As stated above, each class definition containing a call to one of the `META_DEF_x` macros instantiates a metaclass object containing information about that class. Metaclass objects are instances of the class `GnClass`, which may be accessed in the various ways described in the previous sections.

The functions `GnClass::ClassListIterator()))`, `GnClass::IsSubclass()`, and `GnClass::SubclassesIterator()` may only be called after all metaclass objects have been explicitly initialized by calling the function

```
static void GnClass::Initialize().
```

This function must be called at most once during the execution of a program, although it is save to call it not. Clients may call the function

```
static int GnClass::IsInitialized()
```

to check whether the metaclass objects have been initialized. All functions relying on initialized metaclass objects therefore have the following precondition:

```
REQUIRE( GnClass::IsInitialized(), "Metaclass objects must be initialized");
```

GINA++ applications automatically initialize the metaclass objects when the first instance of the class `GnApplication` (or one of its subclasses) is created, which is usually done at the beginning of the `main()` routine. Applications not using the framework classes should therefore initialize the metaclass objects at the beginning of their `main()` routine.

3.5.1 Class Attributes

The following functions return basic properties of classes.

`int IsClassType()`
: returns `1` if the metaclass object is associated with a class type, and `0` otherwise.

`int IsSimpleType()`
: returns `1` if the metaclass object is associated with a non-class type such as `int` or `char`, and `0` otherwise.

The functions `IsClassType()` and `IsSimpleType()` are complementary, i.e., the expression `IsClassType() && IsSimpleType()` is always false.

3.5.2 Class Lists and Iterators

GINA++ defines the class `GnClassList` as a subclass of `GnLinkedList` that may contain pointers to instances of `GnClass` or its subclasses. The class `GnClassListIterator` is defined as a list iterator for lists of class `GnClassList`. Lists and list iterators are discussed in detail in chapter 4.

3.5.3 The List of all Metaclass Objects

The list containing pointers to the metaclass objects of all classes in an application is accessible through `GnClass`:

`static GnClassListIterator *ClassListIterator()`
: This function returns an iterator to the list of all metaclass objects in a running application. It is the clients responsibility to delete this iterator after using it.

For example, the following piece of code prints the names of all classes in an application:

```
GnClassListIterator *iterator = GnClass::ClassListIterator();
for( iterator->Start(); ! iterator->Off(); iterator->Forth() )
    cout << iterator->Item()->Name() << endl;
delete iterator;
```

3.5.4 Querying Base Classes and Subclasses

Each metaclass object stores lists of type `GnClassList` of its direct base classes and subclasses. The following functions return iterators to these lists.

`GnClassListIterator *BaseClassesIterator()`
: returns an iterator to the list of base classes of a class.

`GnClassListIterator *SubclassesIterator()`
: returns an iterator to the list of subclasses of a class.

`int IsSubclass(GnClass *a_class)`
: returns `1` if `this` class is a subclass of or equal to `a_class`, and `0` otherwise. This function is very efficient, and has complexity $O(n)$, where n is the number of direct base classes.

For example, the following function recursively prints the names of all subclasses of a class:

```
void PrintSubclasses(GnClass *a_class)
{
    GnClassListIterator *subclasses = a_class.SubclassesIterator();

    cout << a_class->Name() << endl;
    for( subclasses->Start(); ! subclasses->Off(); subclasses->Forth() )
        PrintSubclasses( subclasses->Item() );
    delete subclasses;
}
```

4 Generic Linked Lists and Iterators

4.1 Introduction

Efficient linked list data structures and iterators are realized by the classes `GnDoubleLinkedList` and `GnDoubleLinkedListIterator`. The list data structure is implemented as a double linked circular chain, providing the base for the implementation of efficient operations for list traversal as well as for inserting and removing elements. Iterators provide efficient operations for traversing over the elements of a linked list.

The classes `GnDoubleLinkedList` and `GnDoubleLinkedListIterator` are not instantiable and serve as templates for concrete list and iterator classes. Specialized list and iterator classes for specific element types must be derived using the techniques described in the next section. An application typically derives one list and one iterator class for each class whose instances are stored in linked lists.

The list class stores pointers to objects, using an extra link object for each element in the list. Therefore, list elements do not have to provide data members for holding the link information, and may be stored in more than one list at a time.

For efficiency reasons, lists and iterator are loosely coupled, i.e. lists are not aware of the iterators attached to them. Clients are always responsible for guaranteeing that iterators remain valid when modifying a list. For example, a client must never move an iterator after he has deleted the element the iterator points to.

4.2 Specializing List and Iterator Classes

The declaration of specialized list and iterator classes holding elements of a specific type is supported by preprocessor macros. These macros simulate the template facility which will become available with C++ 3.0.

In the following discussion, we assume that the class `Person` has already been defined. Then the two statements

```
DefGenericListWithIterator(PersonList,Person,PersonListIterator);
ImplGenericListWithIterator(PersonList,PersonListIterator);
```

declare and implement the class `PersonList` as a subclass ob `GnDoubleLinkedList` that stores pointers to instances of `Person` or subclasses of `Person`. They also declare `PersonListIterator` as an iterator class for lists of class `PersonList`.

The naming scheme followed by GINA++ is to build the name of the list class by appending `List` to the name of the element class, and to build the name of the iterator class by appending `Iterator` to the name of the list class.

The call to `DefGenericListWithIterator` is usually following immediately after the element class definition in the `.h` file, so that the element class and the list and iterator classes have the same visibility.

The call to `ImplGenericListWithIterator` will usually appear in the `.C` file of the list element class. Having a `DefGenericListWithIterator` without a matching `ImplGenericListWithIterator` is illegal and may result in error messages about undefined symbols (`__vtbl_*`) when linking the program.

4.3 Separating the Definition of Lists and Iterators

The following macros allow the separate definition of list and iterator classes:

```
DefGenericList(PersonList,Person);
ImplGenericList(PersonList);
```

These macros define a linked list class `PersonList` for elements of class `Person`. The corresponding iterator class `PersonListIterator` may be defined later with the macros

```
DefGenericIterator(PersonList,Person,PersonListIterator);
ImplGenericIterator(PersonListIterator);
```

4.4 List Macros and Templates

The macros described above guarantee an easy adaption of client code to the template facility of C++ 3.0 when it becomes available. The class definitions of `GnDoubleLinkedList` and `GnDoubleLinkedListIterator` would then be changed into template class definitions:

```
template class GnDoubleLinkedList<ElementType> { .... };
template class GnDoubleLinkedListIterator<ListType,ElementType> { .... };
```

The declaration of subclasses using the above macros would be replaced by normal class definitions:

```
class PersonList : public GnDoubleLinkedList<Person> {};
class PersonListIterator : public GnDoubleLinkedListIterator<PersonList,Person> {};
```

Further changes to client code would not be necessary.

4.5 List Functions

The following description assumes that the classes `Person` and `PersonList` have been defined as above. For simplicity, we describe the list functions as they are defined for these classes. This description must be adapted to other specialized list classes by replacing the element type and the list type as appropriate.

Except when stated otherwise, all list operations have complexity $O(1)$. If a function is said to have complexity $O(n)$, then n means the number of elements in the list.

`PersonList()`
: constructs a new person list.

`~PersonList()`
: destructs a person list. It also frees all memory occupied by link objects. It does not delete the objects stored in the list.

`int Empty() const`
: returns `1` if the list is empty and `0` otherwise.

`int Length() const`
: returns the number of elements currently stored in the list.

`void Clear()`
: removes all elements from the list. It does not delete the elements.

`int Member(Person *person) const`
: returns `1` if the object pointed to by `person` is stored in the list, and `0` otherwise. This function has complexity $O(n)$.

`Person *First() const`
`Person *Last() const`
: returns the first and last element of the list, respectively. It is illegal to call these functions when the list is empty.

`void Append(Person *person)`
: inserts `person` as the last element into the list.

`void Prepend(Person *person)`
: inserts `person` as the first element into the list.

`Person *RemoveFirst()`
: removes the first element from the list and returns that element.

`Person *RemoveLast()`
: removes the last element from the list and returns that element.

`Person *Remove(Person *person)`
: removes the first occurence of `person` from the list and returns `person`. This function has complexity $O(n)$.

`void RemoveAllOccurences(Person *person)`
: removes all occurences of `person` from the list. This function has complexity $O(n)$.

4.6 Iterator Functions

The following description assumes that the classes `Person`, `PersonList`, and `PersonListIterator` have been defined as above. All iterator functions have complexity $O(1)$.

`PersonListIterator(PersonList *list)`
`PersonListIterator(PersonList &list)`
: construct an iterator that is attached to the person list pointed to or referenced by `list`. The new iterator points to the first element of the list if the list is not empty.

`PersonListIterator()`
: constructs an iterator that is not attached to a person list. The iterator may be later attached to a list with the `Attach()` function.

`void Attach(PersonList *list)`
: attaches the iterator to `list`. If the list is not empty, then the iterator is moved to its first element.

`void Detach()`
: cuts the connection between the iterator and the list it is attached to.

`int Off()`
: returns `0` if the iterator points to an element of the list it is attached to, and `1` if the iterator has been either moved before the first or behind the last element. In these cases, the iterator is said to be *off* the list.

`void Start()`
: moves the iterator to the first item of the list if the list is not empty. Otherwise, the iterator is off.

`void Finish()`
: moves the iterator to the last item of the list if the list is not empty. Otherwise, the iterator is off.

`void Forth()`
`void Back()`
: moves an iterator to the next or previous element, respectively. It is illegal to call these functions when the iterator is off.

`int IsFirst()`
`int IsLast()`
: returns `1` if the iterator points to the first or last element of the list, respectively, and returns `0` otherwise.

`Person *Item()`
: returns the element the iterator currently points to. It is illegal to call this function when the iterator is off.

`Person *First()`
`Person *Last()`
: returns the first or last elements of the list the iterator is attached to, respectively. It is illegal to call these functions when the list is empty.

`Person *Remove()`
: removes the element the iterator points to, moves the iterator forward, and returns the removed element. It is illegal to call this function when the iterator is off.

`Person *RemoveNext()`
`Person *RemovePrevious()`
: removes the element after or before the element the iterator points to, respectively. The iterator is not moved. It is illegal to call these functions when there is no next or previous element, i.e. when `Forth()` and `Back()` would move the iterator off the list.

`void PutNext(Person *person)`
`void PutPrevious(Person *person)`
: inserts `person` after or before the element the iterator currently points to, respectively. It is illegal to call these functions when the iterator is off.

`void Swap(PersonListIterator *iterator)`
: exchanges the current element with the element pointed to by `iterator`. Both iterators must not be off, and must be attached to the same list.

4.7 Iterating Through Lists

The macro `LIST_ITERATE` supports iteration through linked lists. The call

```
    LIST_ITERATE(person_list,PersonListIterator,pl_it)
        pl_it.Item()->PrintOn(cout);
```

is equivalent to the statement

```
    for( PersonListIterator pl_it(person_list); ! pl_it.Off(); pl_it.Forth() )
        pl_it.Item()->PrintOn(cout);
```

4.8 Examples

The most common use of iterators is to iterate through a list starting with the first element and moving forward until the iterator has moved after the last element:

```
void PrintPersons(PersonList *person_list)
{
    for( PersonListIterator pl_it(person_list); ! pl_it.Off(); pl_it.Forth() )
        pl_it.Item()->PrintOn(cout);
}
```

The following function removes all elements from the list and deletes them:

```
void RemoveAndDelete(PersonList *person_list)
{
    while( ! person_list->Empty() )
        delete person_list->RemoveFirst();
}
```

5 Dependencies and Change Propagation

5.1 Introduction

GINA++ provides the base classes `GnObservable` and `GnObserver` for modelling dependencies between related objects and for propagating changes to dependent objects. The class `GnObservable` models the notifying part and the class `GnObserver` models the dependent part of a dependency relation. Observed objects notify their observers by using an implicit changed-update mechanism.

Objects that want to notify other objects about changes in their internal state must inherit from the class `GnObservable`. Objects that want to be notified about state changes in dependent objects must inherit from `GnObserver` and must register themselves with the observed object.

5.2 Class `GnObservable`

Observable objects inherit from the class GnObservable. Observable objects keep an internal set of dependent objects. An observed object notifies its dependent objects about state changes by calling the `Changed()` function. This function iterates over the set of dependent objects and calls the virtual `Update()` method on any of these objects.

`Observable()`
: Constructs a new observable object.

`~Observable()`
: Unregisters all observers and destructs the object.

`int IsObserved()`
: Returns `1` if the set of observers is not empty and `0` if the set of observers is empty.

`void Changed()`
: Iterates over the set of observers and calls `Observer::Update()` on any observer in the set. The `Observer::Update()` function is only called if a call to the virtual function `Observer::UpdatePossible()` returns a non-zero value.

`protected: GnObserverList *GetObservers()`
: Returns the set of observers that observe this object. Clients may construct an iterator of class `GnObserverListIterator` to iterate over the set of observers.

5.3 Class `GnObserver`

Observers inherit from the base class `GnObserver`. They may register with the observed object either by passing the address of the observed object to the constructor or by calling `Observable::RegisterObserver()` on the observed object.

Observers are notified by their observed objects about state changes by calling the virtual `Update()` functions. Observer classes specialize this function to implement the required update operation.

`GnObserver()`
: Constructs an unbound observer.

`GnObserver(GnObservable *observable)`
: Constructs an observer that observes the object pointed to by `observable`.

`~GnObserver()`
: Unregisters the observer from the observed object and destructs the object.

`Observe(GnObservable *observable)`
: Registers the observer with the observable object pointed to by `observable`.

`StopObservation(GnObservable *observable)`
: Unregisters the observer from the observed object pointed to by `observable`.

`virtual void Update()`
: This function is called by the observed object to announce a state change. Subclasses specialize this function to implement the required update behavior.

`virtual int UpdatePossible()`
: This function is called by the observed object to determine if a call to `Update()` is legal. Subclasses usually do not specialize this function. This function always returns `1` by default.

5.4 Lists and Iterators

The class `GnObserverList` is a subclass of `GnLinkedList` that stores objects of type `GnObserver`. The class `GnObserverListIterator` provides iterators for lists of class `GnObserverList`.

6 Integration of OSF/Motif and X

In addition to the actual functionality of the framework, GINA++ possesses an interface to OSF/Motif and the X Window System. It provides an encapsulation of the functionality of OSF/Motif and X in C++ classes. The essential advantages of this encapsulation are a closer static type check, a more flexible and uniform concept for the attachment of callbacks as well as a uniform object model for the classes of OSF/Motif and C++. Furthermore, the encapsulation provides a framework for implementing new widget classes to be used, for example, for the implementation of complex dialogs as re-usable classes.

The descriptions of this chapter require a sound knowledge of the concepts of the X-toolkit intrinsics and the widget classes.

6.1 Motivation

The OSF/Motif toolkit, the OSF/Motif widgets, the X-toolkit intrinsics and the X-library are available in the form of three libraries and a set of include files. The include files contain the type declarations, functional definitions and preprocessor directives needed for the use of the libraries. They contain conditionalized code which can be used both by C and C++ programs. This solves the basic problem of interfacing OSF/Motif and X Window System in C++ at a low level.

The programming interface thus available to OSF/Motif and X Window System possesses a number of disadvantages resulting, on the one hand, from the concepts of the X-toolkit, on the other, from some characteristics of C++.

- During changing and reading resources, neither a static type check nor a dynamic type check is carried out in C and C++.
- The use of argument vectors for changing and reading the resources is very error-prone. Frequent errors are as follows: exceeding the bounds of the vector, empty fields in the vector and the specification of a wrong vector length upon calling `XtSetValues()` and `XtGetValues()`.
- Reading individual resource values is complicated as it requires the declaration of a temporary variable, the composition of an argument vector and a subsequent call of `XtGetValues()`. This approach is especially complicated if the resource value is needed as an operand in an arithmetic or Boolean expression.
- The inheritance mechanisms of the X-toolkit intrinsics is artificially installed on C as C knows no inheritance. It differs basically from the inheritance mechanisms in

C++. The definition of independent subclasses of the Motif widgets is complicated and labor-intensive. This is especially true if the functionality of the new subclass is only slightly different from that of its superclass.

- The use of member functions as callback functions is not possible with the exception of functions declared as `static` since the first argument of a member function is (internally) always a pointer to a class variable (`this`), the first argument of a callback function, however, must always be of the type `Widget`.
- The callback mechanism of the X-toolkit intrinsics only allows the transfer of exactly one *client_data* argument to a callback function. Experience shows however that, in many cases, more than one *client_data* argument is needed. Therefore we implemented the data transfer to callback functions with the aid of global variables. The approach to an encapsulation of several *client_data* arguments in one structure, leads to intransparent programs due to the additional type definitions and initializations.
- The handling of modal dialogs is complicated since a return to the main loop of the application is required after displaying the dialog box from a callback function. The processing of user actions is done there by means of one or several callback functions. This sequencing leads in many cases to a disintegration of program units logically belonging together.

The mentioned disadvantages can be avoided if the widget classes of OSF/Motif are adequately encapsulated in C++ classes. This will hide the existing concepts and replace them by better ones.

6.2 Encapsulation in C++ classes

The classes of the encapsulation of OSF/Motif and X can roughly be divided into the following categories:

1. Classes which are used for the encapsulation of Motif widgets .
   The following rule of thumb applies: there is a C++ class `Gn`*Class* for each widget class *class* for which there is a C constructor function of the form `XmCreate`*Class*`()` in the Motif toolkit. Furthermore, there are corresponding classes `GnWidget`, `GnComposite`, `GnConstraint` and `GnPrimitive` for the abstract basic classes `Core`, `Composite`, `Constraint` and `XmPrimitive`. These classes cannot be instantiated and are used for the definition of resources and callbacks.

   The class `GnWidget` also defines the functions of the X-toolkit intrinsics operating on widgets as member functions securing the identity of names as far as possible. For example, the C statement

   ```
   Widget w;
   XtRealizeWidget(w);
   ```

   can be realized in GINA++ by the statement

   ```
   GnWidget w;
   w.Realize();
   ```

The functions of the Motif toolkit which operate on subclasses of `Widget` can also be called in GINA++ as member functions for the corresponding subclasses of `GnWidget`.

The member functions of the class `GnWidget` also constitute a framework for the implementation of new widget classes. The functionality of the available classes can be imported and specialized with the aid of inheritance. For the construction of new dialog classes, several widgets can also be combined to obtain a complex interface module.

2. Classes which realize new widget classes of extended functionality on the basis of precreated widget classes.
   These include the classes for implementing menu bars, hierarchical menus and pop-up menus (see chapter 7).
3. The class `GnCallback` which combines the concepts of callbacks, event handlers, protocol callbacks and timer callbacks in a common abstraction.
   This concept also allows the use of member functions which cannot be entered directly in a callback list of a widget because of their implicitly required `this` argument. The use of member functions as callbacks is however urgently necessary in object-oriented systems as this is the only way to prevent conflicts with (object-oriented) principles such as data encapsulation and modularization by class formation.
4. The class `GnToolkit` which contains the inter-application functions of OSF/Motif and the X-toolkit and whose instance contains the global data of a typical Motif application.
   This includes a method `XtAppMainLoop()` for realizing the main loop of an application. The class `GnToolkit` has exactly one instance per application.
5. The class `GnTimer` whose instances realize *timers*.
   They are needed for realizing program actions which recur within a given time and which have to be executed without user input (e.g. digital clocks/watches, automatic recording of user activities etc.).

6.3 Instantiation of Widgets

The construction of the widget hierarchy follows a top-down procedure as in the case of the X-toolkit. For creating a widget, we first must instantiate an empty C++ widget object. Then the resources of the widget can be set. Next, the function `GnWidget::create()` has to be called in order to create the pertinent Motif widget with the previously set attributes. After that, callbacks can be attached.

The following procedures can be used for creating the empty widget object, the procedure described in item 1 being of preference.

1. The widget is created as an object embedded in another (widget) object:

```
class MyClass {
  public:
    MyClass(GnWidget &a_parent);  // constructor
  protected:
```

```
    GnPushButtonGadget gadget;    // embedded widget object
};

MyClass::MyClass(GnWidget &a_parent)
{
    gadget.create(a_parent, "a_gadget");
}
```

This procedure automatically intitializes the embedded widget object upon the initialization of the wrapping object and destroys it also automatically in the case of the destruction of the wrapping object. The embedded object does therefore not require any additional storage management.

2. The widget is created with the operator `new` as a heap object and is assigned to a pointer variable of the corresponding class (or superclass).

```
// Declare variable, create widget on the heap and assign
GnPushButtonGadget *gadget = new GnPushButtonGadget;
gadget->create(a_parent, "my_gadget");

// GnPushButton is the direct superclass of GnPushButtonGagdet;
GnPushButton       *button = new GnPushButtonGadget;
button->create(a_parent, "my_gadget");

// GnWidget is the superclass of all widgets
GnWidget           *widget = new GnPushButtonGagdet;
widget->create(a_parent, "my_gadget");
```

The creation of a widget as a heap object is always appropriate if the widget is to outlast the execution of the current program layer. Since another cross reference to the widget is needed later as a rule, the storage of the pointer on a component of another object suggests itself:

```
class MyClass {
  public:
    MyClass(GnWidget &a_parent);  // constructor function
    ~MyClass();                  // destructor function
  protected:
    GnPushButtonGadget *gadget;   // pointer to widget object
};

MyClass::MyClass(GnWidget &a_parent)
{
    gadget = new GnPushButtonGagdet;
    gadget->create(a_parent, "a_gadget");
}

MyClass::~MyClass()
{
    delete gadget;
}
```

In this context, it is to be noted that upon destroying an object of the class `MyClass`, the destruction of the gadgets has explicitly to be ensured in the destructor function since otherwise the occupied storage capacity is not released.

The use of heap objects and pointers involves two disadvantages: the management of the widget object on the heap and the pointer variable require additional storage capacity which is likely to exceed the storage capacity required for the actual widget object in most cases. Secondly, a storage release for the widget object has to be secured in a destructor function.

3. The widget is declared in a function as a local object:

```
void MyClass::create_a_gadget(GnWidget &a_parent)
{
    GnPushButtonGadget gadget;
    gadget.create(a_parent, "a_gadget");

    // Further statements
}
```

The disadvantage of this procedure is as follows: the local (widget) objects are automatically destroyed before leaving the layer where they have been defined by the C++ run time system, i.e. their destructor function is called. Since a widget object should in general survive the execution time of the layer which creates it, the use of local objects is usually not recommendable.

After the creation of an empty widget, an optional number of its resources can be set (see section 6.5).

Next, the pertinent Motif widget has to be created with the function `GnWidget::create()` which is defined as follows:

```
void create(GnWidget &parent, char * name)                  // 1.
    { create(&parent, name, preferred_managed()); }

void create(GnWidget *parent, char * name)                  // 2.
    { create(parent, name, preferred_managed()); }

void create(GnWidget &parent, char * name, Boolean managed)   // 3.
    { create(&parent, name, managed); }

void create(GnWidget *parent, char * name, Boolean managed);  // 4.
```

As first argument, the function `GnWidget::create()` needs the parent widget which can be transferred both as a pointer and as a reference. The name of the widget (the *resource name*) has to be specified as second argument. It can later be used for setting further resources by means of the resource data base. The two last definitions (3 and 4) need a Boolean value which specifies whether to *manage* the widget automatically as third value. The widget is *managed* automatically if this argument has the value `True`.

The first definition and the second definition call the fourth definition and determine the value of the argument `managed` by calling the function

```
virtual Boolean preferred_managed();
```

This mechanism simplifies the construction of the widget hierarchy since the function `GnWidget::preferred_managed()` usually delivers the "correct" value for the argument `managed` so that an explicit call of the function `GnWidget::manage ()` is not necessary.

6.4 Destruction of Motif Widgets

For most applications, it is not necessary to destroy a widget which is not needed any longer explicitly since this is done automatically by GINA++. The function `GnWidget::destroy()` can be used for destroying the Motif widget belonging to a C++ widget if required. If necessary, the storage allocated for the C++ widget has to be released subsequently by means of the operator `delete`.

6.5 Resources

In GINA++, the access to the resources of a widget is always done by member functions. The concept of the argument vectors from the X-toolkit is completely hidden by GINA++.

For each resource of a widget, there is one function each for reading and modifying the resource value in the corresponding C++ class. For a resource `XmN`*resource* of the type *type*, the function

`void setR_`*resource*`(` *type* `)`

is defined for modifying the value and the function

type `getR_`*resource*`()`

is defined for reading the value. For the resource types `Widget`, `XmFontList` and `Pixmap`, the `setR_`*resource*`()` functions are overloaded (see below). The overloaded functions execute the frequently required type conversions, they create, for example, the representation of the font required (internally) by the widget from an (external) font name.

The resources modified by means of the `setR_`*resource*`()` functions are not changed directly, but they are first buffered internally in an argument vector. This enables us to modify several resources simultaneously. This vector is used for creating a widget with the function `GnWidget::create()` and subsequently it is deleted automatically. Example:

```
// Declare C++ widget object
GnText text;

// Overwrite and buffer defaults of resources
text.setR_value("a text!");
text.setR_columns(9);
text.setR_editMode(XmSINGLE_LINE_EDIT);

// Create widget with corresponding attributes
text.create(parent, "demo text");
```

If one or more resources are to be modified in a widget already created and visible, the function `GnWidget::set_values()` has to be called after calling the corresponding `setR_`*resource*`()`functions.

```
// Redefine current values of resources
text.setR_value("another text!");
text.setR_column(17);
text.setR_editMode(XmMULTI_LINE_EDIT);

// Modify all resources simultanously
text.set_values();
```

Fonts

For all resources `XmN`*resource* of the type `XmFontList`, the `setR_`*resource*`()`functions are overloaded:

```
void setR_resource(XmFontList font_list);
void setR_resource(char * font_name);
```

The first function sets a font resource if the font is already available in an internal representation as an object of the type `XmFontList`. The second function converts a (external) font name into the internal representation as `XmFontList` first. A list of all fonts is obtained by means of the program `xlsfonts` or the Xlib function `XListFonts()`. For reasons of portability, font names should be coded in the program only in exceptional cases since the availability of a font in a system is not always guaranteed. The fonts should rather be set in the *resource database*.

Pixmaps

For all resources *resource* of the type `Pixmap`, the corresponding `setR_`*resource*`()`functions are also overloaded:

```
void setR_resource(Pixmap pixmap);
Pixmap setR_resource(char * pixmap_file);
Pixmap setR_resource(char * bits, int width, int height);
```

The first function sets a pixmap resource if the pixmap is already available in an internal format. The second function loads a bitmap file in the X11 bitmap format and creates a pixmap structure from the data contained in the bitmap file by means of the Motif function `XmGetPixmap()`. The bitmap file can be created, for example, by means of the bitmap editor `bitmap` available under the X Window System.

The third function creates a pixmap structure from a bit vector and two specifications of dimensions. The bitmap has to be available as a bitmap file in the X11 bitmap format. This format stores the contents of a bitmap as C-code and can be integrated into a C-program by means of a `#include` directive:

```
#include <picture>

// contents of the file "picture"
// #define picture_width 16
// #define picture_height 16
// char picture_bits[] = { 0x12, ....... };
```

```
label.setR_labelPixmap(picture_bits, picture_width, picture_height);
label.setR_labelType(XmPIXMAP);
label.set_values();
```

Widgets

For all resources *resource* of the type `widget`, the pertinent `setR_`*resource*`()` and `getR_`*resource*`()` functions work with the corresponding C++ objects and are defined as follows:

```
void setR_resource(GnWidget *widget);
void setR_resource(GnWidget &widget);
void GnWidget *getR_resource();
```

6.6 Callbacks

Callbacks are implemented in GINA++ with the aid of so-called callback objects which are instances of the class `GnCallback`. These objects contain a pointer to a callback function as well as their arguments. The callback mechanism of GINA++ provides a uniform programming interface to different, though similar callback mechanisms of the X-toolkit, such as event handlers and timers. Furthermore, this concept enables the users to implement their own callback lists in specialized widget classes (see section 6.8).

In the GINA++- widget classes, there are the following two functions for each callback `XmN`*callback*:

```
void add_callback(GnCallback *callback_obj);
void remove_callback(GnCallback *callback_obj, Boolean free_it =True) ;
```

Example: For the class `GnPushButton`, there are the following two functions for the callback `XmNactivateCallback`:

```
void add_activateCallback(GnCallback *callback_obj);
void remove_activateCallback(GnCallback *callback_obj, Boolean free_it = True);
```

6.6.1 Attachment of Callbacks

To attach a callback, we must first create a callback object of the class `GnCallback` or of one of its subclasses. GINA++ provides a number of macros for facilitation. They create the callback objects and execute all necessary type conversions. These macros are as follows:

```
// These macros are required for ``normal'' member functionsn
CALLBACK(class, function, object)
CALLBACK1(class, function, object, class_1,client_data_1)
CALLBACK2(class, function, object, class_1,client_data_1,class_2,client_data_2)

// These macros are required for free functions and static member
```

```
// functions.
SIMPLE_CALLBACK(function)
SIMPLE_CALLBACK1(function, class_1,client_data_1)
SIMPLE_CALLBACK2(function, class_1,client_data_1, class_2,client_data_2)
```

All macros expand to an expression of the type `GnCallback *`. Their use is first demonstrated by some examples in which member functions of the class `Test` are attached to a widget of the class `GnPushButton` as callbacks.

```
class Test {
  public:
    Test(GnWidget *parent_for_button)  // constructor

  protected:
    // member functions to be attached as callbacks

    static StaticMember(caddr_t call_data);
    static StaticMemberWithClientData(void *client_data, caddr_t call_data);

    void OrdinaryMember(caddr_t call_data);
    void OrdinaryMemberWithClientData(void *client_data, caddr_t call_data);
    void OrdinaryMemberWithTwoClientData(void *cld_1, void *cld_2, caddr_t cd);

  private:
    // widget, the callbacks will be attached to
    GnPushButton button;
}
```

The class `Test` possesses five member functions which are to be attached to the C++ widget `button` as a callback. Due to implementation of C++, the last three functions possess a hidden first argument, the so-called `this` pointer. The two member functions defined as `static`, however, do not possess any `this` pointer. When constructing the corresponding callback objects, the programmer has to consider that by using the `SIMPLE_CALLBACK(...)` macros for static member functions and the `CALLBACK(...)` macros for "normal" member functions. This is demonstrated in the code for the constructor function of the class `Test`.

```
Test::Test(GnWidget &parent_for_button)
{
    button.create(parent_for_button, "a_button");

    button.add_activateCallback(SIMPLE_CALLBACK(Test::StaticMember));

    // The GnPushButton is delivered as client_data
    button.add_armCallback(SIMPLE_CALLBACK1(Test::StaticMemberWithClientData,
                                            Test *, this));

    // from hereon the hidden this-pointer must be given as a parameter

    button.add_disarmCallback(CALLBACK(Test, OrdinaryMember, this));

    // In this case, the this-Pointer is delivered as client_data
    button.add_helpCallback(CALLBACK1(Test, OrdinaryMemberWithClientData,
```

```
                                       this,Test *, this));

    // In this case, two client_data are delivered
    button.add_helpCallback(CALLBACK2(Test, OrdinaryMemberWithTwoClientData,
                                      this,Test *,this,GnPushButton *,button));
}
```

In the case of the `SIMPLE_CALLBACK...()` macros, the name of the callback function is always transferred as first argument. The further arguments describe in pairs the *client_data* arguments of the callback function. In the case of the `CALLBACK...()` macros, the first three arguments describe the callback function to be called as well as the object to which the function is to be applied. The remaining arguments describe again in pairs the *client_data* arguments. The meaning of the individual arguments is discussed in section 6.6.3.

6.6.2 Declaration of Callback Functions

Each callback function has to be declared in such a way that its last argument is of the type `caddr_t`. On this argument, the *call_data* argument of the Motif widget to which the callback is attached is transferred to the callback function. The argument has to be converted subsequently into the type delivered by the Motif widget if it is needed in the callback function. This is usually the type `XmAnyCallbackStruct *` or a type derived from it. For details see the documentation of the corresponding OSF/Motif widget classes. In the case of the specialized widget classes of GINA++ (e.g. `GnOptionGroup`), the *call_data* argument is usually a pointer to an instance of the class `GnObject` or a subclass.

The callback macros always convert the *client_data* arguments to the type `caddr_t *`. The callback function should therefore be declared in such a way that its *client_data* arguments are also of the type `caddr_t *`. Within the callback function, the *client-data* arguments have then to be converted back to the "correct" type. It is always the task of the programmer to secure the correct type conversions in a callback function such as shown by the example of the function `OrdinaryMemberWithTwoClientData`:

```
void Test::OrdinaryMemberWithTwoClientData(caddr_t *client_data_1,
                                           caddr_t *client_data_2,
                                           caddr_t call_data)
{
    Test *test          = (Test *)client_data_1;
    GnPushButton *button = (GnPushButton *)client_data_2;

    // further statements
}
```

The callback function can also be declared such that its arguments are already of the "correct" type. This avoids the statements for explicit type conversion, but may lead to problems on machines with different sizes for data types like `int, long` etc.

```
void Test::OrdinaryMemberWithTwoClientData(Test        * client_data_1,
                                           GnPushButton * client_data_2,
                                           caddr_t call_data)
```

```
{
    // Further statements
}
```

6.6.3 Arguments of Callback Macros

In the case of the `SIMPLE_CALLBACK...()` macros, the callback function is specified as first argument in the notation *class*`::`*function*. The description of a *client_data* argument consists of the class of the argument which is expected in the callback function and the actual value of the argument. On this basis, the macros generate statements which execute the necessary type conversion. These are necessary as the class `GnCallback` works internally with the general pointer type `void *` and, therefore, a *client_data* argument has to be converted to the type expected by the callback function immediately upon the construction of a callback object.[1]

The `CALLBACK...()` macros are used for attaching non-static member functions as callbacks. The first three arguments are used for specifying the class in which the callback function is defined, the callback function and the object to be transferred as a `this` pointer to this function. The object has to be transferred to the macro as a pointer; where appropriate, the address of object variables has first to be computed by means of the operator `&`. The specification of the class is necessary to allow the automatic execution of type checks and type conversions. For example, the expression

`CALLBACK(`*class, function, object*`)`

checks that the function *class*`::`*function* is defined and that the type conversion needed for the construction of the callback object

`(void *)(`*class* `*)(`*object*`)`

is feasible.

6.6.4 Specific Subclasses of `GnCallback`

The programmer can form specific subclasses from the class `GnCallback` and integrate them easily into the callback mechanism. The definition of a subclass is sensible, for example, if a callback function requires more than two *client_data* arguments.

6.7 Dialog Boxes

In addition to the dialog classes of OSF/Motif, GINA++ contains a number of specialized dialog classes. They have been derived from the dialog classes of OSF/Motif and complete their functionality.

[1] In contrast to C (and other programming languages), type conversions in C++ are not necessarily no-ops. Therefore the two expressions `(GnWidget *)(void *) button` and `(GnWidget *)button` can deliver different addresses for a pointer variable `button` of the type `GnPushButton *`. This effect occurs especially in the case of multiple inheritance and can lead to bugs which are difficult to localize. Therefore GINA++ requires the specification of the class for callback macros to generate thus a "secure" type conversion of the form `(GnWidget *)(void *)(GnWidget *)button`.

The common functionality of all GINA++ dialog classes is defined in the abstract superclass `GnDialogMixin` and is therefore available in all dialog classes by inheritance. This class defines the functions `pop_up()` and `pop_down()` which are used for displaying and hiding dialog boxes. The call of `pop_down()` is however only required if the resource `XmNautoUnmanage` has the value `False` (default `True`).

6.7.1 Modal (Blocking) Dialogs

The Motif model for handling dialog sequences secures that a dialog box is first displayed in a callback function (`pop_up()`) and that the control is subsequently returned to the main loop of the application. The processing of further user actions is carried from there by further callback functions which are attached to the individual components of the dialog box. This dialog model usually leads to "disintegrated" programs since the code used for processing a dialog sequence is distributed to many, small callback functions. The processing of a (modal) dialog without return to the main loop is very complicated with the concepts of the X-toolkit. The implementation of frequently occurring dialog sequences such as

"save" $\longrightarrow$ "file selection" $\longrightarrow$ "save + error handling"

requires much effort.

For an easier handling of a dialog without return to the main loop, the class `GnDialogMixin` provides the function `alert()` which displays, processes and subsequently deletes a dialog box. The function delivers a result of the type `DialogResult` which can be used for identifying the user action terminating the dialog. A value of the type `DialogResult` can be transferred to the function as an argument. This value is returned as a functional value if the dialog box is hidden by selecting the "Close" menu entry in the menu of the window manager.

For the precreated dialog classes, GINA++ provides further functions for blocking dialog processing (see sections 6.7.2 and 6.7.3).

6.7.2 The Class `GnMessageDialog`

The following descriptions and examples also apply to the classes `GnErrorDialog`, `GnInformationDialog`, `GnQuestionDialog`, `GnWarningDialog`, `GnWorkingDialog`. When importing the examples, replace the string `Message` in the function names and class names by the corresponding part of the desired dialog class (e.g. `GnMessageDialog` $\longrightarrow$ `GnErrorDialog`).

The (free) function `message_dialog()` generates a dialog of the class `GnMessageDialog`, displays it as a modal dialog and waits for the user to answer the dialog. Example:

```
void quit(GnWidget &parent)
{
    if( message_dialog(parent, "Save documents first ?") == OK )
        save_all_documents();
}
```

The function `message_dialog()` delivers a value of the type `DialogResult` as a result. This type is defined as follows.

```
enum DialogResult { OK, Cancel, Yes, No, Help, Select };
```

The functional value depends on the user's answer to the dialog. Depending on the button pressed by the user, i.e. *Ok*, *Cancel* or *Help*, one of the values `OK`, `Cancel` or `Help` is returned. Moreover, the function delivers the value `OK` if the dialog is terminated by double click in the decoration of the window manager. The function `message_dialog()` is declared as follows:

```
DialogResult message_dialog(GnWidget &parent,
                            char * message,
                            char * title = "Message !",
                            unsigned char default_button =
                              XmDIALOG_OK_BUTTON,
                            unsigned char dialog_style =
                              XmDIALOG_FULL_APPLICATION_MODAL);
```

The parameters have the following meaning:

`parent` specifies the parent in the widget hierarchy. The dialog is centered over the middle of its parent.

`message` specifies the text which is to be located in the dialog box.

`title` specifies the title of the dialog the window manager displays in its decoration.

`default_button` identifies a button in the dialog box which can be activated by the return *return* key and which terminates the dialog. This parameter determines the value of the resource `XmNdefaultButtonType` of the class `XmMessageBox`.

`dialog_style` corresponds to the resource `XmNdialogStyle` of the class `XmMessageBox`. The values `XmDIALOG_MODELESS` and `XmDIALOG_WORK_AREA` however are not allowed.

6.7.3 The Class `GnFileSelectionDialog`

The class `GnFileSelectionDialog` encapsulates the Motif class `XmFileSelectionBox`. It provides the following additional functionality:

- The specification of the initial search path and the access to the selected file are done by instances of the class `GnPathname`.
- The selection of a file can be restricted to files of specific characteristics (e.g. to files of specific postfixes (suffixes) or to existing data only etc.).
- The selected file is checked for the satisfaction of specific requirements, e.g. whether it is writable. In the case of error, the user is informed by a corresponding dialog.

The functions `select_new_file()`, `select_old_file()` and `select_old_or_new_file()` facilitate the selection of a file name. The function `select_old_file()` is used for selecting an already existing file. Using `select_new_file()`, we can select a file name which is not yet available in the file system. The function also checks whether the user is allowed to create the file according to his/her read/write authorization. The function `select_old_or_new_file()` selects an optional file name. The three functions deliver the value `OK` if the selection of a file was successful with the constraints given by the parameters. Example:

```
void open(GnWidget &parent)
{
    GnPathname *file;

    if( select_old_file(parent, &file) == OK )
        read_document(file);
}
```

The functional value depends on the users answers to the dialog. Depending on the button pressed by the user, i.e. *Ok*, *Cancel* or *Help*, the function returns one of the values `OK`, `Cancel` or `Help`. Moreover, the function delivers the value `Cancel` if the dialog is terminated by selecting the "Close" menu entry in the menu of the window manager.

The functions `select_new_file()`, `select_old_file()` and `select_old_or_new_file()` possess identical signatures. The meaning of their arguments is illustrated by the example of `select_new_file()` described:

```
DialogResult select_new_file(GnWidget &parent,
                             GnPathname **out_path,
                             char *in_path     = "",
                             char *in_type     = "",
                             char *in_dir      = "",
                             int access        = O_RDONLY,
                             char *title       = "Select a file",
                             time_t after_date = 0,
                             unsigned char dialog_style =
                                 XmDIALOG_FULL_APPLICATION_MODAL);
```

`parent` specifies the parent in the widget hierarchy. The dialog is centered over the middle of this parent.

`outpath` returns the path of the selected file. For this purpose, the address of an available pointer variable of the class `GnPathname` has to be transferred.

`in_path` specifies the name of the file to be provided as a preselection.

`in_type` specifies a regular expression used for restricting the selection to specific files. For example, `"*.[ch]"` provides only those files for selection which show the postfixes (suffixes) `.c` or `.h`.

`in_dir` specifies the directory in which files are to be selected. The default is the *current working directory.*

`access` specifies the access rights required for the selected file. After selection, the function checks whether the file can be opened with the corresponding attributes. Admissible values are all possible combinations of the values `O_RDONLY`, `O_WRONLY` and `O_RDWR` with bit-by-bit OR.

`title` specifies the title of the dialog in the decoration of the window manager.

`after_date` specifies a time mark (in milliseconds since UNIX hour 0) after which the creation of the selected file has to be terminated.

`dialog_style` corresponds to the resource `XmNdialogStyle` of the class `XmMessageBox`. The values `XmDIALOG_MODELESS` and `XmDIALOG_WORK_AREA` are however not allowed.

6.8 Implementation of Specific Widget Classes

GINA++ provides a framework for the implementation of new widget classes. The power of this framework does not cover at all the functionality of the X-toolkit intrinsics for the implementation of new widget classes since the two systems are based on different object models. The framework is rather to be used to combine parts of the user interface, which belong together from a functional viewpoint, to a new class.

We discuss the implementation of a new widget class by means of the example of the GINA++- class `GnHistoryScroller`. In GINA++, this class implements the so-called *history scroller* which is displayed in a GINA++ application after the selection of the menu entry "History Scroller".

An instance of the class `GnHistoryScroller` is a non-modal dialog box which contains two *push button widgets*, one *label widget* and one *scroll bar widget* as further components. The two push buttons allow a gradual undo and redo and display the name of the next command in the dialog history as a text. The label widget displays the number of the commands in the undo list. The scroll bar widget is used for a quick scrolling of the undo and redo lists.

We use a *form widget* for grouping the dialog components. For grouping the dialog components, we also use a *row column widget* which includes the two push button widgets and the label widget. For optical reasons, the label widget is provided with a frame which is realized with a frame widget.

When describing the definition of a new widget class, we concentrate on the parts which are used for the creation of the dialog components, the setting of the resource fields and the attachment of the callbacks. We discuss the interconnection of application and dialog and the parts used for optimization as far as this is absolutely necessary for understanding.

6.8.1 Class Definition

The following presents the complete specification of the class `GnHistoryScroller`. The code is stored in the file `Gina/GnHistoryScroller.h` and has to be integrated into the program by means of an `#include` directive as soon as the class is used.

```
class GnHistoryScroller : public GnFormDialog {
  public:
    // Constructor
    GnHistoryScroller ( GnDocument *doc );
```

```
    // Accessor and modifier for 'busy' flag
    void Busy(Boolean state) { busy = state; };
    Boolean Busy()           { return(busy); };

    // Display new contents of undo/redo history
    void update();

  protected:
    // Inherited from base class
    virtual void before_create (GnWidget *);
    virtual void after_create ();

    // Callback handlers
    void scroll_history(caddr_t call_data);
    void change_mapped(caddr_t call_data);

    // Dialog components
    GnPushButton previous_button, next_button;
    GnLabel      cmd_num_label;
    GnScrollBar  scroll_bar;
    GnRowColumn  column;
    GnFrame      frame;

    // Associated document object
    GnDocument   *document;

    // For optimization and flicker-free redisplay
    Boolean      popped_up;
    Boolean      busy;
    int          old_undo_length;
    int          old_redo_length;
};
```

As a basic class, we choose the class `GnFormDialog` which obtains the general characteristics of a dialog box and the functionality of the *form widget* by multiple inheritance. The class `GnFormDialog` is defined in the file `Gina/GnFormD.h` which is integrated by means of an `#include` directive.

The constructor function transfers a pointer to the document which the history scroller is assigned to. This reference allows the access to the lists of the command objects stored in the document object. The dialog components are implemented by embedded objects (see section 6.3). The required class definitions are also integrated with `#include` directives.

The composition of the individual dialog components is done in the two virtual functions `before_create()` and `after_create()` which are obtained by the basic class via inheritance. The function `before_create()` is called by GINA++ directly before creating the main widget. In our case, the main widget is the dialog box with the form widget. In the function `before_create ()`, the resources of the dialog box can be set, for example. The function `after_create ()` is called by GINA++ directly after the creation of the main widget. It creates the components of the dialog box, sets the resource fields of the latter and attaches the callbacks.

Furthermore, the class definition contains the two functions `scroll_history()` and `change_mapped()` which are attached to the dialog components as callbacks. The function `update()` is called by the document object when the lists of the command objects are changed. The private members `popped_up`, `busy`, `old_undo_length` and `old_redo_length` are used for internal purposes. The function `Busy()` is used as an access function for the member `busy`.

6.8.2 Instantiation

The following example shows how to create a history scroller widget. The history scroller is assigned to the document and is stored in the document object on the member variable `history_scroller`.

```
#include <Gina/GnHistoryScroller.h>

void GnDocument::CreateHistoryScroller()
{
    history_scroller = new GnHistoryScroller(this);
    history_scroller->create(shell, "HistoryScroller");
}
```

The operator `new` is used for creating a new object. The `this` pointer, i.e. a reference to the document object, is transferred to the constructor function. The embedded widget objects are initialized in the constructor. The Motif widgets associated with the widget objects are however created later in the function `after_create()`.

The function `GnWidget::create()` creates the dialog box with the inner dialog components. It calls the two virtual functions `GnHistoryScroller::before_create()` and `GnHistoryScroller::after_create()` which are defined for the first time in the class `GnWidget` and which are specialized in the class `GnHistoryScroller`.

6.8.3 Implementation

Only the members are initialized in the constructor. The creation of the dialog components is done later in the function `after_create`.

```
GnHistoryScroller ::
GnHistoryScroller ( GnDocument *doc )
{
    document        = doc;
    popped_up       = False;
    busy            = False;
    old_undo_length = old_redo_length = -1;
}
```

The resource fields of the form widget are set in the function `before_create()`. At this moment, there is still no Motif widget so that callbacks can be attached only later.

```
void GnHistoryScroller ::
before_create(GnWidget *)
```

```
{
    setR_dialogTitle(GnXMSTRING_LTOR("History Scroller"));
    setR_noResize(True);
}
```

The function `after_create()` creates all dialog components. In this context, the `this` pointer can be used for referring to the form widget as a parent widget. When calling the function `after_create()`, the dialog components have already been initialized by the constructor, but their resource fields have still to be set and the Motif widgets have to be created by means of the function `GnWidget::create()`. At this time, the hierarchy of the widgets is also defined.

Two callbacks are attached to the form widget for optimization purposes. They register the visibility state of the dialog box on the member `popped_up` and secure that modifications to the dialog are only carried out when the dialog is visible. The function `scroll_history()` is attached to the scroll bar widget as a callback which informs the history scroller of any manipulation by the user and which secures the execution of *Undo* and *Redo*. Callbacks which secure the execution of *Undo* and *Redo* are also attached to the two push button widgets. Since only a single command is repeated or canceled here, the functions provided for this purpose in the class `GnDocument` can be attached directly.

```
void GnHistoryScroller ::
after_create ()
{
    add_unmapCallback(CALLBACK(GnHistoryScroller, change_mapped, this));
    add_mapCallback(CALLBACK(GnHistoryScroller, change_mapped, this));

    scroll_bar.setR_orientation( XmHORIZONTAL );
    scroll_bar.setR_minimum(1);
    scroll_bar.setR_maximum(2);
    scroll_bar.setR_sliderSize(1);
    scroll_bar.setR_value(1);
    scroll_bar.setR_rightAttachment(XmATTACH_FORM);
    scroll_bar.setR_topAttachment(XmATTACH_FORM);
    scroll_bar.setR_leftAttachment(XmATTACH_FORM);

    scroll_bar.create(this, "historyScrollBar" );

    scroll_bar.add_valueChangedCallback(CALLBACK(GnHistoryScroller,
                                            scroll_history, this));
    scroll_bar.add_dragCallback(CALLBACK(GnHistoryScroller,
                                    scroll_history, this));

    column.setR_marginWidth(0);
    column.setR_orientation(XmHORIZONTAL);
    column.setR_topAttachment(XmATTACH_WIDGET);
    column.setR_topWidget(scroll_bar);
    column.setR_rightAttachment(XmATTACH_FORM);
    column.setR_leftAttachment(XmATTACH_FORM);
    column.setR_bottomAttachment(XmATTACH_FORM);
    column.create(this, "historyButtonRow");
```

```
    previous_button.setR_recomputeSize(False);
    previous_button.create(column, "historyUndoButton" );
    previous_button.add_activateCallback(CALLBACK(GnDocument,
                                              receive_undo_callback,
                                              document));
    frame.create(column, frame");

    cmd_num_label.setR_recomputeSize(False);
    cmd_num_label.create(frame, "historyLabel");

    next_button.setR_recomputeSize(False);
    next_button.create(column, "historyNextButton");
    next_button.add_activateCallback(CALLBACK(GnDocument,
                                          receive_redo_callback,
                                          document));
}
```

The function `change_mapped()` supervises the visibility status of the dialog box. For this purpose, it analyzes the contents of the `call_data` argument and registers the state on the member `popped_up`. In the case of a modification of the state from invisible to visible, the function `update()` is called. It adjusts the dialog to the current contents of the undo and redo lists.

```
void GnHistoryScroller ::
change_mapped(caddr_t call_data)
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *)call_data;

    Require(cbs->reason == XmCR_MAP || cbs->reason == XmCR_UNMAP );

    if( cbs->reason == XmCR_MAP ) {
        popped_up = True;
        update();
        document->history_scroller_menu_entry->set_sensitive(False);
    } else {
        popped_up = False;
        document->history_scroller_menu_entry->set_sensitive(True);
    }
}
```

6.8.4 Summary

The definition of a new widget class is in accordance with the following scheme:

- The specification of the class is recorded in a specific `.h` file. The file contents have to be protected against multiple loading by preprocessor directives of the form:

```
#ifndef _GnHistoryScroller_H_
#define _GnHistoryScroller_H_

// class definitions etc.

#endif _GnHistoryScroller_H_
```

- First, the basic classes have to be defined. A multiple inheritance from the class `GnDialogMixin` has to be secured for dialog classes. In most cases, this class is inherited indirectly via one of the available dialog classes (e.g. in the case of the class `GnFormDialog`).
- The definitions of all used classes (regardless of whether they are used as basic classes or as embedded objects) have to be included.
- In general, the functions `before_create()` and `after_create()` have to be redefined. In some cases, the definition of a constructor is necessary.
- If possible, the dialog components should be designed as embedded objects in the dialog object. This secures a compact code and also constitutes an efficient solution.
- The constructor does not yet set resources and creates no dialog components.
- The function `before_creae()` sets resource fields of the dialog box.
- The function `after_create()` creates the dialog components and their resource fields and attaches callbacks.
- Callbacks of the dialog components are usually sent first to the dialog box, and the dialog box transfers them to the outside world.

7 Menu System

7.1 Introduction

GINA++ offers an easy interface to the OSF/Motif menu system. It supports pulldown as well as popup menus . All standard menu entries offered by Motif — including cascading menus — may be used; furthermore, GINA++ supports some menu entries not immediatly availiable in Motif, like toggle groups.

All menu entries are subclasses of the class `GnMenuEntry`, which, in turn, is a subclass of `GnObserver` (see chapter 5). This allows the easy implementation of update mechanisms for individual entries by overriding the `Update()` function.

A menu entry object is different from the Motif widget used to display the entry. This is important for two reasons: first, it allows menus to be dynamically rearranged, i.e. entries can be inserted or deleted at runtime. Second, it allows the sharing of menu entries: the same entry can be used in different menus (e.g. a pulldown and a popup menu).

Each entry maintains a list of associated widgets. Every time an entry is inserted into a menu, a new widget is created and added to the list. It is possible to add a widget to the list that is not part of the menu hierarchy, e.g. a pushbutton on a dialog box. In this case, the button is controlled by the entry since each change of the entry (e.g. of its name) is reflected by all associated widgets.

7.2 Standard Entries

GINA++ supports a variety of standard menu entries. The different classes and the main functions availiable to the programmer are described below.

Please note that functions already described for a superclass are not repeated for each subclass.

The base class for all menu entries is the class `GnMenuEntry`. It is intended as a pure base class which offers most of the functionality of every menu entry; it is not advisable to actually instantiate this class.

The most important functions deal with the settings of various parameters of the entry:

```
GnMenuEntry(char *label, GnCallback *callback = NULL,
       Boolean sensitive = True, char *accelerator = NULL,
       char *modifier = NULL, char mnemonic = 0)
```

This creates a menu entry with the name `label`. This name can be used later to identify the entry; in most cases it is also used as the resource name of the associated widgets. All other parameters are optional: `accelerator` and `modifier` specify the accelerator key to be used. If `accelerator` is `NULL`, no accelerator key is installed; if `modifier` is `NULL`, `''Alt''` is used as default modifier key.

`mnemonic` specifies which mnemonic to use; if it is 0, no mnemonic is installed. For best results, the mnemonic character should appear in the name of the entry. `callback` specifies the function to call when the entry is activated. `sensitive` decides whether the entry is sensitive to mouse input. Please note that this function does *not* create any widgets for the entry; this is done when the entry is inserted into a menu.

`~GnMenuEntry()`
: Deletes the menu entry; all associated widgets are destroyed and deleted, too.

`char *GetName()`
: Returns the name of the menu entry.

`Boolean GetSensitive()`
: Returns whether the menu entry is sensitive to mouse input.

`char GetMnemonic()`
: Returns the mnemonic character of the menu entry.

`GnWidget *GetWidget(GnWidget *parent = NULL)`
: Returns the first widget from the list of associated widgets of the menu entry. This function should be used if the entry is inserted in only *one* menu; in this case there will be only one associated widget in the list.

`EntryWidgetList *GetWidgetList()`
: This function returns the list of all widgets associated with the menu entry. See also the description of the macro `SET_ENTRY_WIDGETS()` below.

`SET_ENTRY_WIDGETS(obj, type, funccall)`
: This macro can be used when a special function is to be called for all widgets associated with a menu entry. The parameters of the macro are a pointer to the menu entry object `obj`, the type of the widgets of that entry (`type`) and the function, including parameters, that is to be called (`funccall`). As an example, the macro call SET_ENTRY_WIDGETS(my_entry, GnToggleButton, setR_indicatorOn(False)) would disable the indicator display for all widgets associated with the toggle entry my_entry.

`void SetSensitive(Boolean sensitive)`
: Sets whether the menu entry is sensitive to mouse input or not.

`void SetMnemonic(char mnemonic)`
: Sets the mnemonic character for the menu entry; 0 stands for no mnemonic.

`void SetAccelerator(char *accelerator, char *modifier = NULL)`
: Sets the accelerator key for the menu entry. If `accelerator` is `NULL`, no accelerator is installed. The `modifier` parameter specifies which modifier key to use; it defaults to `''Alt''` if it is `NULL`.

`GnWidget *CreateWidget(GnWidget *parent, char *name = NULL)`
: This function creates a widget for the menu entry with the given parent and resource name, and sets all resources — callback, label, etc. — according to the specification of the entry. If an entry is only to appear inside of menus, this function does not need to be called explicitly. It can be called, however, to install a widget outside of a menu that is to be controlled by the menu entry.

`void DestroyWidget(GnWidget *parent)`
: Destroys the widget associated with the menu entry with the given parent.

`GnSeparatorEntry` is a subclass of `GnMenuEntry` which describes a simple separator line in a menu. There is only one important additional function for this class: the constructor. Please note that objects of this class ignore the settings of accelerators, mnemonics and callbacks since they do not react to mouse input, anyway.

`GnSeparatorEntry()`
: Creates a new separator entry.

Another menu entry that ignores the settings of accelerators, mnemonics and callbacks because it does not react to mouse input is described by the class `GnLabelEntry`. These entries can be used as menu titles etc. Again, the only important function for this class is the constructor:

`GnLabelEntry(char *label)`
: Creates a new label entry.

Possibly the most important subclass of `GnMenuEntry` is the class `GnTextEntry`. It describes a standard – pushbutton – menu entry .

```
GnTextEntry(char *label, GnCallback *callback = NULL,
      Boolean sensitive = True, char *accelerator = NULL,
      char *modifier = NULL, char mnemonic = 0)
```
: Creates a new text entry. The parameters are the same as for the constructor of `GnMenuEntry`.

Another important class of menu entries are toggle entries. These can be used to indicate a boolean value by displaying an indicator flag besides the label text.

```
GnToggleEntry(char *label, Boolean value = True,
      GnCallback *callback = NULL, Boolean sensitive = True,
      char *accelerator = NULL, char *modifier = NULL, char mnemonic = 0)
```
: Creates a new toggle entry. The internal state is set to the value given by the parameter `value`. All other parameters are the same as for the constructor of `GnMenuEntry`.

`Boolean GetState()`
: Returns the internal state of the toggle entry.

`void SetState(Boolean)`
: Sets the internal state of the toggle entry to the given value.

7.3 Group Entries

GINA++ supports a special kind of menu entry called *group entry*. There is no equivalent Motif widget. Group entries allow 1-of-many (*radio group*) or n-of-many selections (*toggle group*) by displaying a list of toggle buttons with the appropriate behaviour.

Since these groups are used in a similiar way in the `GnSelection` classes, `GnRadioGroupEntry` is a subclass of both `GnGroupEntry` and `GnSingleSelection`; `GnToggleGroupEntry` is a subclass of `GnGroupEntry` and `GnMultipleSelection`. The additional functions for group entries besides the functions inherited from `GnMenuEntry` and `GnSelection` are merely the constructors and destructors:

`GnGroupEntry(GnCallback *callback = NULL, Boolean sensitive = True)`
: Creates a new group entry. For a description of the parameters, see the discussion of `GnMenuEntry` above. Please note that the callback is activated if *any* of the group buttons is pressed.

`~GnGroupEntry()`
: Deletes a group entry. All associated widgets are destroyed and deleted, too.

`GnRadioGroupEntry(GnCallback *callback = NULL, Boolean sensitive = True)`
: Create a new radio group entry. For a description of the parameters, see the discussion of `GnMenuEntry` above.

`GnToggleGroupEntry(GnCallback * = NULL, Boolean = True)`
: Creates a new toggle group entry. For a description of the parameters, see the discussion of `GnMenuEntry` above.

The code fragment below demonstrates the usage of a group entry. First, a new radio group entry is created. The callback that is to be activated if *any* button of the group is changed is given in the constructor. The second step consists in adding the entries of the radio group (named "demo 1-3" in the example) by using the `add()` function inherited from `GnSelection`. Each entry will be displayed as a separate button when the complete menu is displayed. Finally, the group entry is added to the desired menu — in this case to the standard `Edit` menu of the main menu.

```
      ⋮

   GnRadioGroupEntry *demogroup = new GnRadioGroupEntry(
                    CALLBACK(DemoDocument, Demofunc, this), True);
   demogroup->add("demo 1");
   demogroup->add("demo 2");
   demogroup->add("demo 3");
   GetMainShell()->GetMainMenu()->FindEntry("Edit")->AddEntry(demogroup);

      ⋮
```

Below is a short example how a callback function could be implemented. In this example, the name of a button is printed if it is pressed.

```
void DemoDocument::Demofunc(caddr_t call_data)
{
    // demo callback function: print name of selected item

    char buf[32];

    GnSingleSelection *sel = (GnSingleSelection *)call_data;
    cout << sel->SelectedItem()->PrintOn(buf,32);
}
```

7.4 Menus

Menus are a special kind of menu entries, and so the class `GnMenu` is a subclass of `GnMenuEntry`. They may be used in the same way as other entries. In particular, they may be inserted into other menus, allowing the construction of cascading menus of arbitrary depth.

A menu object can be used as a pulldown menu of the main menu (by inserting it into the main menu), as a submenu of another menu (by inserting it into that menu) or as a popup menu (by registering it with the widget on which it should appear with the function `CreatePopup()`). This must be done before adding any entries to it.

The most important additional functions deal with the inserting and removing of entries to a menu:

`GnMenu(Boolean sensitive = True, char mnemonic = 0)`
: Creates a new menu. The parameter `sensitive` decides whether the menu should be sensible to mouse input — this is only used if the menu is used as a pulldown menu. `mnemonic` specifies which mnemonic to use; if it is 0, no mnemonic is installed.

`~GnMenu()`
: Deletes the menu and all entries that have been added to it. All associated Motif widgets are destroyed, too.

`void AddEntry(GnMenuEntry *entry, Boolean separator = False)`
: Adds the menu entry described by the parameter `entry` to the menu. If the optional parameter `separator` is `True`, a separator line is inserted before the entry. This function can not be used to insert submenus into a menu; see the description of the function `AddEntry()` below.

`void AddEntry(GnMenu *entry, char *title, Boolean separator = False)`
: Adds the submenu described by the parameter `entry` to the menu. The title used for the submenu is given by the parameter `title`. If the optional parameter `separator` is `True`, a separator line is inserted before the entry. Please note that the effects of inserting a menu into itself or into a submenu of itself are undefined.

`void InsertEntry(GnMenuEntry *entry, GnMenuEntry *before = NULL, Boolean separator = False)`
: Inserts the menu entry described by the parameter `entry` into the menu. The entry

is inserted before the entry described by the parameter `before`. If `before` is not part of the menu or is `NULL`, the new entry is simply added at the end of the menu. If the optional parameter `separator` is `True`, a separator line is inserted before the new entry.

`GnMenuEntry *RemoveEntry(GnMenuEntry *entry)`
: Removes the menu entry described by the parameter `entry` from the menu and returns a pointer to it. If the entry is not part of the menu, a `NULL` pointer is returned.

`GnMenuEntry *FindEntry(char *item)`
: Returns a pointer to the menu entry with the name given by the parameter `item`. If no such entry exists, a `NULL` pointer is returned.

`void CreatePopup(GnWidget *parent)`
: If a menu is to be used as a popup menu, this function has to be called to register it with the parent given by the parameter `parent` before any entries are added to it. Please note that a menu can be used both as a pulldown menu — by adding it to another menu — and as a popup menu at the same time. In the same way, a menu can be used as popup menu for different parents by calling `CreatePopup()` for each of them.

`void Popup(GnWidget *parent, XEvent *event)`
: This function pops up a menu previously installed with `CreatePopup()` on the given parent. The second parameter describes the X event which is necessary to obtain the position of the menu on the screen. This parameter is most easily obtained by a call of `GnApplication::GetLastEvent()`.

7.5 Main Menu

The class `GnMainMenu` differs from the previously described class `GnMenu`: it is not a subclass of `GnMenuEntry` but of `GnMenuBar`, so all functions to manipulate the associated widget may be used directly.

The class offers functions to insert, remove or find complete pulldown menus described by a `GnMenu` object. There is also a set of convenience functions which allow to insert (remove, find) single text or toggle entries to a certain topic of the main menu, creating the topic if necessary.

The most important functions are:

```
void AddEntry(GnMenu *menu, char *topic, char *wname,
     Boolean help = False)
```
: Adds the menu described by the parameter `menu` to the main menu under the given topic. The name specified by the parameter `wname` is used as resource name of the associated Motif widgets. The optional parameter `help` indicates whether this menu is intended as a help menu; if it is `True`, the menu is placed at the right end of the menu bar. If the menu topic is “Help” or “help”, it is automatically recognized as a help menu.

`void AddEntry(GnMenu *menu, char *topic, Boolean help = False)`

The only difference to the function described before is that the name given for the menu is used as resource name for the associated Motif widgets.

`void RemoveEntry(GnMenu *menu)`

The menu described by the parameter `menu` is removed from the main menu if present; if not, this function has no effect.

`void RemoveEntry(char *topic)`

The menu with the given topic is removed from the main menu, if the topic is present; if not, this function has no effect. The strings are only compared up to the length of the specified topic, so short forms may be used.

`GnMenu *FindEntry(char *topic)`

A pointer to the menu with the given topic is returned, if the topic is present in the main menu. Otherwise, this function returns a `NULL` pointer.

```
GnMenuEntry *AddMenuCommand(char *topic, char *wtopic, char *item,
      char *witem, GnCallback *callback, Boolean sensitive = True,
      int options = MNEMONIC, char *accelerator = NULL,
      char *modifier = NULL, char mnemonic = 0)
```

This function adds a text entry or a toggle entry with the name given by `item` to the menu with the topic given by `topic` and returns a pointer to the newly created entry. If there is no menu for the specified topic, a new one is created. The names specified by the parameters `wtopic` and `witem` are used as resource names for the associated Motif widgets of the menu and the menu entry.

All optional parameters are described in the function `InsertMenuCommand()` below.

```
GnMenuEntry *AddMenuCommand(char *topic, char *item,
      GnCallback *callback, Boolean sensitive = True,
      int options = MNEMONIC, char *accelerator = NULL,
      char *modifier = NULL, char mnemonic = 0)
```

The only difference to the function described before is that the resource names for the associated Motif widgets of the menu and the entry are taken from the names of the menu and the entry, resp.

```
GnMenuEntry *InsertMenuCommand(char *topic, char *wtopic, char *item,
      char *witem, char *before, GnCallback *callback,
      Boolean sensitive = True, int options = MNEMONIC,
      char *accelerator = NULL, char *modifier = NULL, char mnemonic = 0)
```

This function inserts a text or a toggle entry with the name given by `item` to the menu with the topic given by `topic` and returns a pointer to the newly created entry. If there is no menu for the specified topic, a new one is created. The new entry is placed before the entry with the name given by the parameter `before`. The names specified by the parameters `wtopic` and `witem` are used as resource names of the associated Motif widgets of the menu and the menu entry.

There are lots of optional parameters: `sensitive` decides whether the entry should be sensitive to mouse input. `options` is supposed to contain a bitmask describing the features of the new entry: possible are combinations of the constants M_MNEMONIC, M_ACCELERATOR, M_SEPARATOR and M_TOGGLE. If M_MNEMONIC is present the new entry uses a mnemonic; if M_ACCELERATOR is present, it uses an accelerator, too. If M_SEPARATOR is present, a separator entry is created in the menu before the entry to be inserted. Finally, if M_TOGGLE is present, a toggle entry is created instead of a text entry.

If M_ACCELERATOR is used, a accelerator key and an optional modifier key description must be specified in the parameters `accelerator` and `modifier`. For a discussion of the format of these strings see above.

If M_MNEMONIC is used, a memonic letter may be specified in the parameter `mnemonic`; if it is 0, the mnemonic letter is choosen automatically by the system.

```
GnMenuEntry *InsertMenuCommand(char *topic, char *item, char *before,
    GnCallback *callback, Boolean sensitive = True,
    int options = MNEMONIC, char *accelerator = NULL,
    char *modifier = NULL, char mnemonic = 0)
```

The only difference to the function described before is that the resource names for the associated Motif widgets of the menu and the entry are taken from the names of the menu and the entry, resp.

```
GnMenuEntry *RemoveMenuCommand(char *topic, char *item)
```

This function removes the menu entry with the name specified by the parameter `item` from the menu with the topic given by `topic` and returns a pointer to it. Please note that even if the last entry from a menu is removed, the menu itself and the topic will remain. If the specified entry does not exist, a `NULL` pointer is returned.

```
GnMenuEntry *FindMenuCommand(char *topic, char *item)
```

A pointer to the menu entry with the name specified by the parameter `item` in the menu with the given topic is returned, if it exists. Otherwise, a `NULL` pointer is returned. The strings are only compared up to the length of the specified topic resp. item, so short forms may be used.

```
GnSeparatorEntry *AddSeparator(char *topic)
```

A separator is added to the menu with the given topic.

8 View and View Objects

Views can be used to present parts of the contents of a document using graphics. They provide both a *procedural* and an *object oriented* interface to graphical output as well as an interface for direct manipulation interaction techniques. The procedural interface consists of a set of basic drawing functions and maintains default graphic contexts as well as default fonts. The object oriented interface is built on top of the procedural one and provides support for graphical *view objects* such as rectangles, circles and bitmap icons. View objects maintain their own size and position, can be moved and resized by the application program and are redrawn by GINA++ whenever necessary. The programmer of an application may indicate, that view objects are interactively *selectable*, *movable* and/or *resizable* by the user, by simply setting the value of specific flags. *Direct manipulation object* is a subclass of view objects, that supports all the facilities mentioned above by default.

8.1 The View

The class GnView is a subclass of GnDrawingArea, which encapsulates the Motif class XmDrawingArea . The Ressources of that class are inherited and may be set using the corresponding `setR_ressource()` calls. Typically views are created and integrated into the widget hierarchie within the method `CreateWindows()` of the class GnDocument or a subclass respectively. A view has a X Window to draw into and maintains a default graphics context for graphical operations, a default graphics context for text and a special graphics context to be used to draw feedback of interactive operations. A view has a list of installed view objects, that are automatically redrawn if necessary. Mouse operations of the user within the visible part of a view are reported to the function `button_press()` of the view. Key strokes are reported to `key_press()` of a view if the related widget has the input focus.

We explain the most important functions exported by the class `GnView` below.

`GnView(GnDocument *doc)`
: A pointer to the document to be presented by the view is passed as a parameter to the constructor of the view. Note that you can not do initializations that depend on the X Window being already created in the constructor of your subclasses of `GnView`. The window is created later. See `draw()` below.

`GnDocument* get_document()`
: This function returns the pointer to the document to be presented by the view.

`Boolean DrawingIsPossible()`
Drawing is not possible immediately after the view is created, because the widget might not be created yet. This is indicated by the return value of this function.

`Window get_x_window()`
Returns the window id of the related X Window. This can be used in calls to Xlib functions that are not directly supported by GINA++.

`virtual void determine_window_attributes()`
Various window attributes are set to default values within this function, e.g. the graphic contexts and the window id. You should avoid overwriting this function or be very carefull. Look at the body of the function in the source code.

`GC get_gcontext()`
Returns the default graphics context . This can be used in calls to Xlib functions that are not directly supported by GINA++, e.g. when changing attributes of the context.

`GC get_feedback_gc()`
Returns the feedback context. This can again be used in calls to Xlib functions that are not directly supported by GINA++, especially when changing attributes of the context.

`GnViewObjectList *get_view_objects()`
Returns a pointer to the list of currently installed view objects.

`Boolean overlapping_objects()`
Should indicate whether there are overlapping view objects installed in the view. A return value of `False` can be used to optimize redrawing of the view objects. Currently returns always `True`.

`GnWidget *scroller()`
Returns a pointer to the parent widget of the view. This will typically be a scroller.

`Boolean PropagateButtonPress()`
If this function returns `True`, mouse events will be reported to the first object in the list of installed view object, that is “hit” by the event, if any. This is decided by calling the function `point_inside()` of each installed object, see below.

`void PropagateButtonPress(Boolean propagate)`
Can be used to determine whether mouse events should be propagated to an installed view object if hit. If the passed parameter evaluates to `True`, events will be propagated.

`virtual void button_press(int code, int repetition, int x, int y)`
This function is called by the framework after the user has pressed down a mouse button within the window of the view and `PropagateButtonPress()` returns `False` or no view object is hit. The mouse position is passed in the parameters `x` and `y`. Parameter `repetition` contains the number of user generated mouse events within a time intervall specified by the variable `double_click_intervall`. The parameter

`code` reports the mouse and keyboard key the user combined in producing the event. The possible values are `SELECT` for Button1, `EXTEND` for Button1 + Shift, `TOGGLE` for Button1 + Ctrl, `INSERT` for Button2, `MENU` for Button3 and `OTHER` for any other combination. The function will be overwritten in most subclasses of GnView and will typically start different mouse down commands (see 9.3) depending on the state of the application.

`virtual void insert_selection(int x, int y)`
This function is called if a mouse event with code `INSERT` has been generated by the user and no view object has been hit.

`virtual void key_press(char key, int code, unsigned int modifiers, int x, int y)`
This function is called by the framework after the user has pressed a keyboard key while the view has the input focus. Nothing is done by default. Note that accelarator key combinations will also be reported to the view if it has the input focus. The parameters reported are the corresponding entries in the `XKeyEvent` structure generated by the X Window System.

`virtual void key_release(char key, int code, unsigned int modifiers, int x, int y)`
Is called after the user released the pressed key. See above.

`Boolean unselect_all_objects()`
Unselects all selected view object installed in the view. The return value indicates whether there where selected objects.

`void clear_area(int x, int y, Dimension width, Dimension height)`
The rectangle specified by the parameters is cleared in the window. An expose event will be generated for the area, so that `draw()` (see below) will be called.

`void force_redraw()`
This function may be called, whenever the content of the document has changed and a complete redraw of the view seems to be necessary. The window is cleared and `draw()` will be called.

`void invalidate_rectangle(int, int, Dimension, Dimension)`
This function allows an optimization when the document has changed and it can be determined that only a part of the view has to be redrawn to correctly reflect the contents of the document.

`virtual void draw(int count, int x, int y, Dimension width, Dimension height)`
By overwriting `draw()` in subclasses, the specific presentational behaviour of the view can be determined. This function is called by the framework whenever the window has to be redrawn to correctly present the contents of the document. This may occur after an *expose event* generated by the X Server if parts of the window become visible on the screen, after internal calls to `force_redraw()` or `invalidate_rectangle()`. Note that installed view objects are maintained automatically by the framework (see `after_draw()`). So `draw()` can be used to draw a special background or when view

objects seem to be to complex to be used in solving a simple presentation problem. Note also that initializations, that require the X Window already to be created, can not be done in the constructor of the view, because the window is created after the creation of the view object. `draw` can be used to do those initializations. A boolean member variable of the view object (e.g named `first_expose_event`) can be used to indicate whether `draw()` is been called for the first time. This will be in response to the first expose event, when the window is created, so initializations can be done depending on the value of the variable within `draw()`. Parameter `count` tells the number of times `draw()` will be called for different regions on reaction of expose events of the X Server. It can be used for optimization. The other parameters specify the rectangle of the view to be drawn. The function does nothing by default.

`virtual void after_draw(int count, int x, int y, Dimension width, Dimension height)`
This function is called after a call to `draw()` has been executed. Its main purpose is to redraw the installed view objects by calling their member function `draw()`. It will be seldom specialized in subclasses of GnView, e.g to change the order in which the view objects are drawn (default drawing order is the order of installation) or if automatic redraw is not wished. Look at the source code before you overwrite the function.

`virtual void resized()`
This function will be called by the framework whenever the view has been resized, e.g. by internal calls to `setR_width()` or `setR_height()`. It allows for rescaling of sizes and positions of view objects etc. and does nothing by default.

The following functions are customizations of the corresponding Xlib Calls. These calls require parameters determining the display, the drawable and the graphic context to be used. This parameters are taken from default values within GINA++ and therefore do not need to be specified. Refer to the Xlib Manual for details. A parameter `fill` always indicates whether a geometric object to be drawn should be filled using the same color as for the border.

`void draw_point(int x, int y)`

`void draw_points(XPoint *points, int npoints, Boolean relative=FALSE)`
If parameter `relative` is `True`, the coordinates of point *N* are interpreted relative to the coordinates of point *N - 1*. If it is `False`, the coordinates of all points are interpreted relative to the origin of the coordinate system.

`void draw_line(int x1, int y1, int x2, int y2, Boolean relative =FALSE)`
If parameter `relative` is `True`, the coordinates of the second point are interpretated relative to the coordinates of the first point. If it is `False`, the coordinates of all points are interpreted relative to the origin of the coordinate system.

`void draw_lines(XPoint *points, int npoints, Boolean relative =FALSE, Boolean fill =FALSE, int shape =Complex)`
Parameter `relative` has the same semantic as in `draw_points()`. If `fill` is `True`,

the polygon will be filled using `XFillPolygon()`. Parameter `shape` is required to have value `Complex`, `Convex` or `Nonconvex`. If you do not know whether your polygon is convex or self-intersecting, pass `Complex`, which will allways be save, but maybe slow.

```
void draw_segments(XSegment *segments, int nsegments)
```

```
void draw_rectangle(int x, int y, Dimension width, Dimension height,
    Boolean fill =FALSE)
```

```
void draw_rectangles(XRectangle *rects, int nrects, Boolean fill =FALSE)
```

```
void draw_arc(int x, int y, Dimension width, Dimension height,
      int angle1, int angle2, Boolean fill =FALSE)
```

```
void draw_arcs(XArc *arcs, int narcs, Boolean fill =FALSE)
```

```
void draw_glyph(int x, int y, char c)
```

```
void draw_glyphs(int x, int y, char* str)
```

8.2 View Objects

A view object maintains a graphical object to be displayed in a view. It has a position, an enclosing rectangle, determines the shape of the graphical object and covers an area of the view, which might be smaller than the enclosing rectangle, but should never exceed it.

A view object may support different interaction styles: It may be selectable, movable and/or resizable. View objects do not support any of this *facilities* by default. You may activate any combination of facilities by setting the `Facilities` variable for example in the constructor or elsewhere in subclasses. Subclass `GnDirectManipulationObject` has all facilities enabled.

A view object has a set of graphic context values, which will be used in drawing the object. For every *component* of the Xlib Structure `XGCValues` a public function `setGC_`*component*`()` with the appropriate parameter is defined, e.g. `void setGC_line_width(3);` sets the line width to be used in drawing lines to 3, 3 being of type `int`.

What follows is a brief description of the interface provided by view objects.

```
GnViewObject(XGCValues& cg_val, Dimension wid, Dimension hei,
      Boolean mouse_sens = TRUE)
```

The parameter `cg_val` contains the values of components of the graphics context to be used when drawing the object in the view. The initial size of the object is given by `wid` and `hei`. The position is set to a dummy value using the origin of the coordinate system, because a position will only make sense after the object has been installed into a parent view. The boolean parameter indicates, whether the object is *mouse sensitive*, i.e. whether the parent view should pass a mouse event to the object if it is hit.

`int get_x_pos()`
Returns the x coordinate of the upper left corner of the enclosing rectangle of the graphical object.

`int get_y_pos()`
Returns the y coordinate of the upper left corner of the enclosing rectangle of the graphical object.

`Dimension get_width()`
Returns the width of the enclosing rectangle of the graphical object.

`Dimension get_height()`
Returns the height of the enclosing rectangle of the graphical object.

`GnView * get_parent_view()`
Returns the view the view object is installed within, if any. If the object is not installed, `NULL` will be returned.

`void install(GnView *, Boolean redraw =TRUE)`
The object will be installed in the view given by the first parameter, i.e. will be registered in the views list of view objects to be automatically refreshed if necessary. If `redraw` is `True`, the area of the enclosing rectangle of the object will be redrawn in the view. This description of `redraw` holds for all further occurrences of the parameter within this section. Be carefull not to install an object in another view before deinstalling it, an object installed in more than one view could never be deinstalled from all of them.

`void install(GnView *, int x, int y, Boolean redraw =TRUE)`
The only difference to the previously described function is that the object is moved to position `x, y` while installing it.

`void deinstall (Boolean redraw =TRUE)`
The object is deinstalled from its parent view, i.e removed from the list of view objects of that view. It will no longer be visible, but will not be destroyed, so that it may be reused later.

`void move(int new_x, int new_y, Boolean redraw = TRUE)`
The coordinates of the upper left corner are set to `new_x` and `new_y` respectively.

```
void resize(Dimension new_width, Dimension new_height,
    Boolean redraw =TRUE)
```
The width and height of the enclosing rectangle are set to `new_width` and `new_height`.

`int facilities()`
A view object may support different direct manipulation interactions. These are called facilities. An object may be selectable, movable and/or resizable. The integer value returned by `facilities()` is to be interpreted as a bit-array containing the information about the supported facilities. For a selectable object

`(facilities() & OBJ_SELECTABLE)` evaluates to `True`. For a movable object `(facilities() & OBJ_MOVABLE)` evaluates to `True` and for a resizable object gtt(facilities() & OBJ_RESIZABLE) is `True`. By default a view object has no facilities.

`void select(Boolean redraw = True)`
: This function is called by GINA++ if a view object has been selected by the user with mouse click inside the enclosing rectangle . It may be called from inside the application program, e.g. to restore a previous selection or start a nonempty document with a default selection. Other view objects, that are selected will become unselected, the selected object will be redrawn in a manner, that reflects its new state by using `draw_selected()`. To implement specific behaviour on selection for view objects specialize `became_selected()`.

`void unselect(Boolean redraw = True)`
: This function implements the inverse function of `select()`.
See `became_unselected()` also.

`void toggle_selected(Boolean redraw = True)`
: After a call to this function a selected view object will become unselected and vice versa.

`is_selected()`
: Returns `True` if the view object is selected, `False` otherwise.

`virtual void became_selected()`
: This function is called by GINA++ whenever `select()` is being executed. It can be used to implement specific behaviour on selection in subclasses of `GnViewObject`.

`virtual void became_unselected()`
: This function is called by GINA++ whenever `unselect()` is being executed.

`Boolean is_mouse_sensitive()`
: This function is called by the parent_view to determine if an installed view object should be tested whether it is hit by a users mouse click or not. It returns the value of the member variable `mouse_sens` wich in turn is set in the constructor and may be changed in subclasses.

`Boolean point_inside (int x, int y)`
: This function should return `True`, if the given coordinates describe a point within the area covered by the geometrical object on the screen governed by the view object, otherwise stated, if the object is *hit*. By default the enclosing rectangle is tested, but subclasses should present a refined behaviour. A circle object for example should return `True` only if the point is inside the circle, not if only the enclosing rectangle is hit.

`void invalidate_rectangle(int x, int y, Dimension width, Dimension height)`
: A rectangle of size `width` and `height` located on position `x, y` *relative* to the upper left corner of the enclosing rectangle will be invalidated in the parent view, thereby forcing a redraw of that area.

`void force_redraw()`

The enclosing rectangle of the view object will be invalidated and redrawn in the parent view.

`virtual void draw( int count, int x, int y, Dimension width, Dimension height)`

This function determines the look of the view object on the screen. By default, a rectangle is drawn. The function will be called anytime an area to be redrawn in the parent view intersects with the enclosing rectangle of the view object. The intersecting rectangle to be restored is given by parameters `x, y, width` and `height`. `count` has the same meaning as in the above described function `draw()` of the view.

`virtual void draw_selected(int count, int x, int y, Dimension width, Dimension height)`

This function is called instead of `draw()` if the object is selected. It is used to change the apparence of the object on the screen to indicate its selected state to the user. By default small squares called *handles* are drawn at the corners of the enclosing rectangle of the object.

`Boolean get_outlines()`

During the feedback phase of either a move or a resize operation a feedback is drawn. If `get_outlines()` returns `True`, which is the default, the object itself remains unchanged during this phase and the feedback is given by frequent calls to `draw_outline()`. If `get_outlines()` returns `False`, the object itself is moved or resized multiple times during feedback phase and the feedback is given by drawing the object with its actual postion and size. `reconfigured_by_user()` by user will also be called in this case whenever the object is redrawn.

`virtual void draw_outline(int new_x, int new_y, Dimension width, Dimension height, Boolean clear)`

This function draws a version of the object, that can easily be identified by the user as a feedback. The default behaviour is to draw a rectangular rubberband line beginning at the upper left corner of the enclosing rectangle of the object. It is called frequently during the feedback phase if `get_outlines()` returns `True`.

`int get_handle_size()`

This function determines the size of the handles of a selected object. You can supply a fixed value or a value depending on the size of the object etc. The default return value is 5, meaning that the handle is a square of 5x5 pixels.

`virtual void constrain_position(int & x, int & y)`

If a movable object is hit without hitting the handles, a *mover* is started. This is a command object of type `GnObjectMover`. It will frequently call `constrain_position()`, to allow constraining the upper left corner of the object to move to a subset of all positions in the view. The parameters contain the proposed new position. They may be changed if they do not match the applications needs.

`virtual void constrain_position_and_size(int & x, int & x, Dimension & width, Dimension &height)`

If a resizable object is hit at one of its handles, a *resizer* is started. This is a command object of type `GnObjectResizer`. It will frequently call `constrain_position_and_size()`. The main purpose of this function is to allow the definition of constraints of the size of an object to resize, i.e. its width and height. Some constraints of the size lead to unpleasent behaviour of the object on the screen, if the postion cannot be changed too, so the coordinates of the upper left corner are passed as changable parameters.

`virtual void constrain_size(Dimension & width, Dimension & height)`
This function has been called instead of `constrain_position_and_size()` in earlier versions of GINA++. It is now called within that function by default to maintain upward compatibility.

`virtual void reconfigured_by_user(Boolean moved, Boolean resized)`
Whenever an object has been moved or resized by the user, this function is called, so that the application program can react. The parameters indicate whether an object has been moved and/or a resized.

`virtual void button_press(int code, int repetition, int x, int y)`
If an object is mouse sensitive and has been hit by a mouse click, `button_press()` is called, to pass control over the event from the parent view to the object. The meaning of the parameters is the same as for the function `button_press()` of `GnView` above. Depending on `code` different actions take place. Note that this function already defines a complex behaviour by default, so be carefull to replace it only if inevitably necassary.

`virtual void double_clicked(int code, int repetition, int x, int y )`
This function is called, if the object was hit by more than one mouse click within a specified time intervall. The meaning of the parameters is the same as for the function `button_press()`.

The following functions correspond to the above explained drawing functions of the view. The parameters have the same meaning. The difference is, that all coordinates given by `x, y` are interpreted relative to the upper left corner of the enclosing rectangle of the view object.

```
void draw_point(int x, int y)

void draw_points(XPoint *points, int npoints, Boolean relative =FALSE)

void draw_line(int x1, int y1, int x2, int y2, Boolean relative =FALSE)

void draw_lines(XPoint *points, int npoints, Boolean relative =FALSE,
    Boolean fill =FALSE, int shape =Complex)

void draw_segments(XSegment *segments, int nsegments)

void draw_rectangle(int x, int y, Dimension width, Dimension height,
    Boolean fill =FALSE)
```

```
void draw_rectangles(XRectangle *rectangle, int nrectangles,
      Boolean fill =FALSE)

void draw_arc(int x, int y, Dimension width, Dimension height,
      int angle1, int angle2, Boolean fill =FALSE)

void draw_arcs(XArc *arcs, int narcs, Boolean fill =FALSE)

void draw_glyph(int x, int y, char c)

void draw_glyphs(int x, int y, char *string)
```

9 Unlimited Undo and Command Objects

9.1 Introduction

All applications build with GINA++ may use a general undo and redo mechanism. This facility is presented to the user by a collection of entries within the menu “Edit” of the menu bar of each open document. The user may select “Undo” to reverse the effects of his last command or repeat an undone command by selecting “Redo”. All commands issued by the user form a linear *history* of the document. The user can iterativly undo or redo all commands. Once a command is undone, the previous one becomes undoable, once a command is redone, the next one becomes redoable. The *history scroller* dialog provides push buttons for single undo and redo and a scrollbar for undoing and redoing multiple commands at a time.

All kinds of user actions and interactions are modelled through *command classes*. A command class implements the intended effects of an operation, and optionally implements the inverse operation, building the base for unlimited *undoing* and *redoing*. A *command object* is created for each user action and is submitted to and maintained by GINA++. The data members of command objects hold the state information necessary to undo and redo the associated operation.

Internally the history is represented by two linear lists of command objects within the object of `class GnDocument` representing the document. One of these lists containes objects representing the commands the user issued but did not undo, the other one containes objects representing the commands, that are undone and thus are redoable. The second list will be empty as long as the user has not undone any commands or whenever he issued a new command.

The programmer of a GINA++-Application will not deal directly with this lists. Instead he will encapsulate all user actions that are to be reflected in the history in command objects. Different commands are implemented as subclasses of `GnCommand`. Commands with special interactive behaviour can be implemented as subclasses of `GnMouseDownCommand`.

9.2 Commands

All user actions that cause changes of the contents of the document, should be encapsulated in command objects. The different types of possible user actions are represented in subclasses of `class GnCommand`. Whenever a user action actually takes place, an object of the appropriate class has to be instantiated and has to be *submitted* to the framework by calling the method `submit()` of that object. The constructor used to instantiate the command object needs at least a pointer to an object of `class GnDocument` or subclasses as a parameter. Optional parameters are a pointer to an object of `class GnView` or subclasses and a `Boolean` indicating whether a checkpoint should be created. The pointer to the view determines the view to be used to visualize the effects of the command. Checkpoints are currently not supported.

A submitted command object is registered in the history of the document. To realise the intended effects to the contents of the document, the method `doit()` of the object is called by the framework. The programmer thus has to specialize this method to achieve the desired effects when implementing the subclass. If the user wants to undo or redo a command, the framework calls the method `undoit()` of the corresponding command object or `redoit()` respectively. Again the programmer has to specialize these methods.

Every subclass of `class GnCommand` inherits some virtual methods of type `Boolean` indicating whether a command is `executable()`, is `undoable()`, `causes_change()` to the contents of the document or represents a time consuming modal dialog step to be presented to the user by a special `clock_cursor()`. `executable()` is checked before the command object is submitted to the framework. If a command causes a change and is not undoable, all command objects in the history prior to that object will no longer be undoable. A special clock cursor is not supported in the current release.

It is the programmers responsibility to specialize the mentioned methods in an appropriate way, especially to make `undoit()` the inverse function of `doit()` and mark command objects, that are not undoable by forcing `undoable()` to return `False` in that case. Otherwise the user might destroy the contents of his document by undo or redo operations, that are illegal but still possible. For the same reason it is important to encapsulate **all** user actions in command objects.

The member functions the programmer will deal with are described in detail below.

`GnCommand(GnDocument *, GnView * = NULL, create_checkpoint = FALSE)`
: The first parameter points to the document object the command object will be send to for registration in the history. The second parameter must only be given, if the view to be used for displaying the resulting contents of the document is not the main view. It is then used to automatically scroll to the correct position on undo and redo operations. Checkpoints are currently not supported.

`virtual char *name() = 0`
: Pure virtual function, that must be specialized in subclasses to return a string containing the name to be used in presenting the objects of that class at the user interface, e.g. in the history scroller.

`virtual void submit()`
: This function submits the command object to the framework, i.e. sends it to the document for registration within the history and for execution.

`virtual Boolean executable()`
: Indicates whether the command object is executable within the current state of the application. Default return value is `True`. Command objects are only submitted on a call to `submit()` if `executable()` returns `True`.

`virtual Boolean causes_change()`
Indicates whether the contents of the document is changed by executing the command. If `causes_change()` returns `True` the document is marked modified. If a command causes change to the document and is not undoable all prior commands become also undoable. Default return value is `True`.

`virtual Boolean undoable()`
Indicates whether a command is undoable. Only undoable commands are registered in the history. Default return value is `True`. See `causes_change()` also.

`virtual Boolean clock_cursor()`
A return value of `True` indicates that a special clock cursor should appear on the screen during the execution of the command, because it is a time consuming modal dialog step. Currently not supported, default return value is `False`.

`virtual void doit()`
This function has to be specialized to implement the operation the command is intended to perform. It is called by the framework when the command is submitted.

`virtual void undoit()`
This function is called by the framework on a users request to undo an operation. It must therefore be specialized to implement exactly the inverse operation of the function `doit()`.

`virtual void redoit()`
This function is called by the framework on a users request to redo an operation previously undone. It must therefore implement an operation with the same effects to the contents of the document as `doit`. The difference beween `redoit()` and `doit()` can be used to implement performance enhancements for redo operations, e.g. the result of a time consuming calculation may be saved within the command object (not in the document) when performing `doit()` and maybe used by `redoit()` instead of doing a recalculation. By default `redoit()` calls `doit()`.

`virtual void commit(Boolean undone)`
This function is called by the framework whenever a command object is removed from the history to allow cleanup actions to be performed. The parameter indicates whether the status of the command is undone or done.

`virtual void scroll_before_redo()`
If a command is redone or undone, the part of the view, that reflects the related internal changes should be visible on the screen. Therefore GINA++ does automatic scrolling if necessary. The scrolling behaviour is implemented in the function `scroll_before_redo()` which may be overwritten to meet special needs.

`virtual void scroll_before_undo()`
See `scroll_before_redo()`

9.3 Mouse Down Commands

A user action can often be specified by moving the mouse with a pressed button from one location within a view to another, where the pressed button is released. Objects or a special area might be selected for further operation, the start and end point of a move of a graphical object might be specified or the size of a graphical object to be drawn can be determined in that way. While the mouse is being moved, a graphical feedback is desirable in most cases. In GINA++ user actions of that kind are modelled through *mouse-down-command* classes.

A mouse down command object is typically created from the method `button_press()` of a view or a view object (see chapter 8) and immediately submitted by calling `submit()`. The object is internally registered as the only active mouse down command of the application as long as the user keeps the mouse button pressed. This phase is referred to as the *feedback phase* below. During this phase the functions `constrain_mouse()`, `draw_feedback()` and `track_mouse()` are repeatedly called to make an immediate visible feedback of the operation to the user possible. Special behaviour for phases without mouse movement within feedback phase can be implemented by overwriting the functions `idle_timeout()` and `mouse_idle()`. The feedback phase terminates, when the user releases the pressed button. `call_doit()` is called to decide whether the command should be executed or canceled. In the first case, the object is treated as a normal command object, in the later case `not_submitted()` is called to make clean up operations possible. The object will not longer be registered as active mouse down command in the application object.

Again we describe the important member functions below.

`GnMouseDownCommand( GnDocument *, GnView *, int start_x, int start_y )`
: In contrast to the parent class the pointer to a view is not optional here. The last two parameters are supposed to give the position where the mouse button was pressed.

`virtual void constrain_mouse (int &x, int &y)`
: This function can be used to simplify the users task of giving correct input. It is called for every mouse position. `x` and `y` contain the coordinates of the mouse position and may be changed by the function to conform to specific constraints . If for example a square should be drawn and the actual mouse position specifies a rectangle, the coordinates may be changed to fullfill the conditions of a square. The actual cursor position will not be changed, but the values to be used internally as the mouse position are different. The effect will be visible immediately through the drawn feedback. There are no constraints implemented by default.

`virtual void draw_feedback(int x, int y, Boolean clear)`
: This function is called repeatedly during the feedback phase, to draw a feedback. For every position of the mouse cursor, the function is called two times. At the first call parameter `clear` is set to value `False`, to indicate that a feedback should be drawn, at the second call `clear` is set to `True`, to indicate that the feedback should be cleared and the area on the screen previously occupied by the feedback has to be restored to its former state. By default a rectangular rubberband line is drawn, where the starting point of the command is the upper left corner and the mouse position is the right lower corner of the rectangle.

`virtual void track_mouse(int x, int y, Boolean started, Boolean finished)`

This function is similar to `draw_feedback()`. It will be called once for every mouse cursor position. The first call is indicated by `started` being `True` and `finished` being `False`. The last call comes with the two parameters having reversed values. On all calls in between, the two parameters have the value `False`.

`virtual void not_submitted()`

If the command object will not be submitted to the framework for execution and registration when the user releases the mouse button as a result of various conditions (see below), this function is called to make clean up actions possible.

`virtual int hysteresis()`

This function returns a value of minimum movement of the mouse in either direction measured in pixels. If the movement was less, `doit()` will not be called and the command object will not be registered in the history. Default return value is 2.

`virtual Boolean auto_scrolling()`

A return value of `True` indicates that scrolling is to be done if the mouse cursor leaves the visible part of the view. This is the default. The operation of automatic scrolling is currently not performed by GINA++.

`virtual int idle_timeout()`

Specifies a time intervall to wait after the last movement of the mouse, before `mouse_idle()` will be called to perform a specific idle action. The default return value is 250 milliseconds.

`virtual void mouse_idle (int x, int)`

This function is called, if the time intervall specified by `idle_timeout()` has expired since the last movement of the mouse. The parameters contain the coordinates of the mouse cursor. No operation is implemented by default.

`virtual Boolean call_doit()`

If the user releases the button and has previously moved the mouse cursor more than `hysteresis()` pixels away from the start position, `call_doit()` is called to determine if the command should be executed. Arbitrary evaluations of the command object and the status of the document may take place within the body of the function. The default return value is simply `True`.

10 Example of Constructing an Application

This chapter describes how to develop an application with GINA++. It does not present all facilities and details of GINA++, but rather shows the most important classes and functions and their use by means of an example, a simple graphics editor. You can use this application as a model for your own applications and add extensions and further features of GINA++ gradually.

Section 10.1 describes what our application is supposed to do, and discusses which functionality is provided by GINA++, what has to be added by the programmer and how it has to be done. Section 10.2 gives an overview of the classes that make up the graphics editor. Section 10.3 discusses the implementation of these classes in detail.

A summarized version of the source code of all classes can be found in appendix A. The source code is also contained in the GINA++ distribution in machine-readable form (see appendix A).

10.1 Specification and Realization

The purpose of the application is to manage and manipulate *graphical documents*. A graphical document consist of *graphical objects*, such as rectangles, squares, ellipses, circles, and triangles as well as text. Document management includes operations like creating new documents, loading existing documents, and saving and printing of documents. Documents are manipulated by adding, removing, resizing, and rearranging graphical objects. All modifications to a document should be done by direct manipulation using a mouse, with an individual graphical feedback for each type of operation. Additionally, all modifications to a document shall be undoable and redoable. The actual type of graphical objects that will be drawn shall be selectable through a *palette*, which contains an icon for each object type. The main window of the graphics editor is shown in figure 10.1

The basic functionality for managing documents is available through GINA++. Access to these functionality is possible by using a shell widget of class `GnDocumentShell` for the applications main window. This shell contains a menu bar with entries “Create”, “Open”, “Save”, “Print”, and “Close”. What is left to the programmer is to derive subclasses of `GnApplication` and `GnDocument` x and to redefine a few small virtual functions which parameterize these operations.

The menu bar also contains entries for undoing and redoing of command stored in the command history of the document. The programmer’s task is to derive subclasses of `GnCommand` and `GnMouseDownCommand` which model the operations for modifying the contents of a document and handle individual graphical feedback.

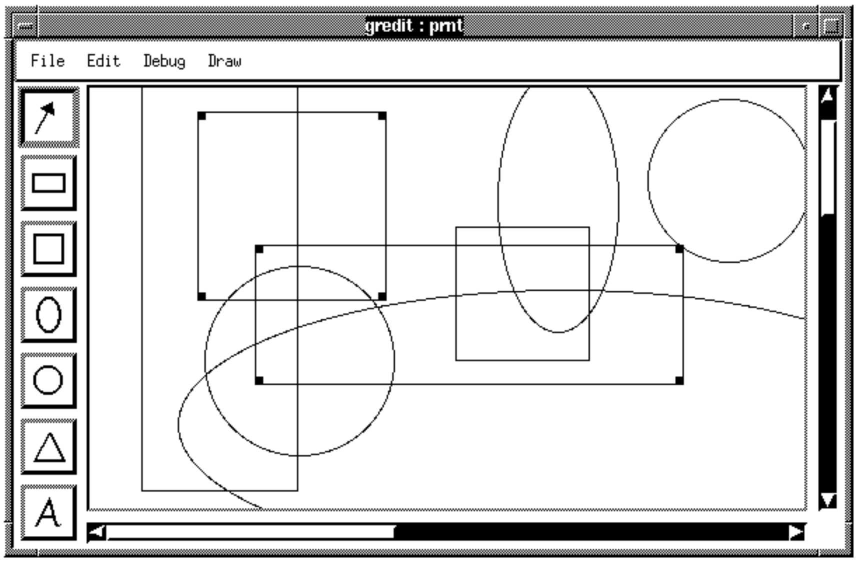

Figure 10.1: Main Window of the Graphics Editor

The class `GnView` serves as a working area for displaying and manipulating graphical objects and provides aids like graphics contexts for drawing graphical feedbacks. The programmer must derive a subclass of `GnView` that translates mouse and keyboard input into operations by creating and submitting appropriate command objects.

The different types of direct-manipulatable graphical objects must be modelled by subclasses of `GnDirectManipulationObject`. Rectangular and elliptical objects are already provided by the classes `GnRectangleO` and `GnEllipseO`.

10.2 The Class Hierarchy

The classes that make up the graphics editor application are shown in figure 10.2. Classes inside ovals belong to GINA++. The classes inside rectangles have to be implemented for the graphics editor . In the following we give a brief overview on these classes.

`GraphicEditorApplication`
: determines properties of the application, such as the document type, its signature, and the file type.

`GraphicEditorDocument`
: specifies the contents of a graphical document and the structure of the widget tree.

`GraphicEditorView`
: maps mouse and keyboard input to application specific commands.

`ViewObjectDrawer`
: is an abstract base class for commands that draw graphical objects with individual feedback.

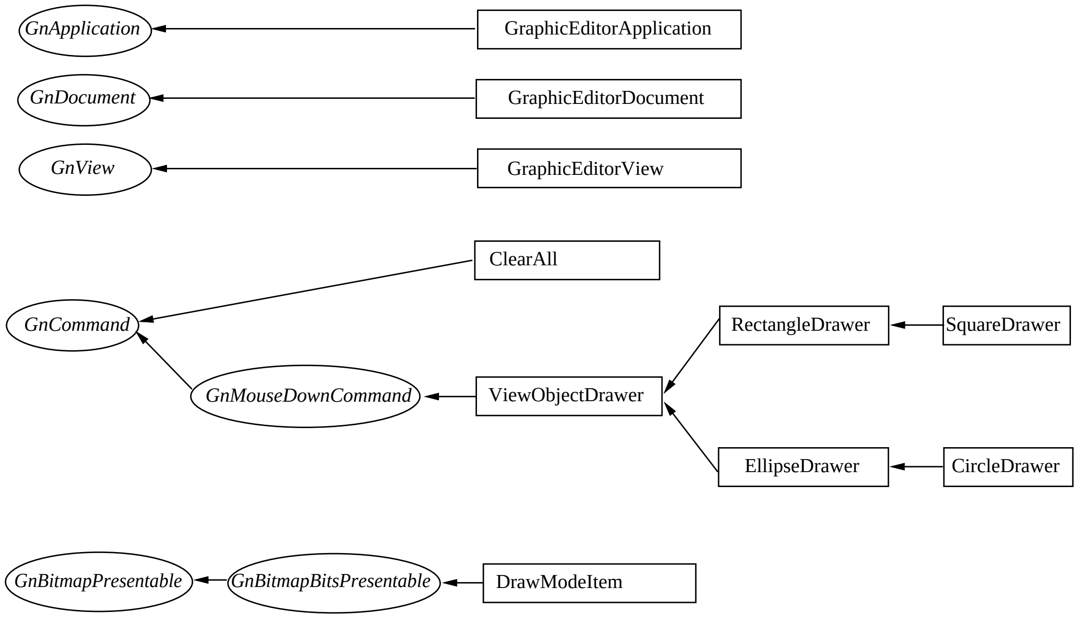

Figure 10.2: Class Hierarchy of the Graphics Editor

`RectangleDrawer`, `SquareDrawer`, `EllipseDrawer`, `CircleDrawer`
: are concrete classes for commands that draw rectangles, squares, ellipses, and circles, respectively.

`ClearAll`
: is a command class for deleting all graphical objects in a document.

`DrawModeItem` and `enum DrawMode`
: The type `DrawMode` determines the possible states of the palette. Instances of the class `DrawModeItem` associate a bitmap icon with each possible state of the palette.

10.3 Class Definitions

This section discusses the specification and implementation of all classes we need for the graphics editor application. We start off with a discussion of the classes that must be implemented in almost any application in order to fit the new application into the framework, and continue with the classes that implement the application-specific behavior.

All classes uses the macros `META_DEF` and `META_IMPL` to define the functions necessary for integrating them with the meta-object protocol (see chapter 3 and section 3.2). These macros generate a constructor function for internal use for each class. It is therefore necessary that classes implement an empty default constructor when required, because the C++ does not generate one when other constructors are present. We leave out the discussion of these constructors, and describe only the non-trivial ones.

10.3.1 The `main` routine

The purpose of the `main()` routine is to create an instance of the class `GraphicEditorApplication`, to call its `run()` function and to pass in the command line options contained in `argv`.

```
#include <Gina/Gina.h>
// include application-specific .h and .C files here

int main( unsigned int argc, char ** argv )
{
    GraphicEditorApplication gredit;

    gredit.run( argc, argv );
}
```

The source file containing the `main()` routine also includes all class definitions of GINA++ as well as all source files of the graphics editor application (see section A).

10.3.2 Class `GraphicEditorApplication`

The development for a new application always begins with the definition of a subclass of `GnApplication`. We call this new application class `GraphicEditorApplication`:

```
class GraphicEditorApplication : public GnApplication {
    META_DEF_1(GraphicEditorApplication,GnApplication);
  public:
    GraphicEditorApplication();
  protected:
    virtual GnDocument *CreateDocument();
  protected:
    virtual char *AppClass ()    { return "Gredit" ; }
    virtual char *AppSignature() { return "gredit" ; }
    virtual char *AppFileType()  { return ".gredit" ; }
};
```

The specialized virtual functions are indispensable for a proper functioning of the application, therefore they have to be specialized in each application. All functions are protected, so they may only be called indirectly from other framework functions, i.e. after calling `gredit.run()` from the `main()` routine.

The virtual function `CreateDocument()` creates and returns an instance of the application-specific subclass of `GnDocument`. This function is called by the framework to create a new document of the application-specific type, for example when the menu entry “New” has been activated.

```
GnDocument *GraphicEditorApplication ::
CreateDocument ()
{
    return new GraphicEditorDocument;
}
```

The function AppClass() delivers the name of the application class the application belongs to. This name may be used as the application's class name when specifying X resource values through the resource data base of the X window system (Xdefaults). The function AppSignature() specifies the *signature* of an application. It is written to any document the application creates, and may be used by other applications (e.g. a graphical shell) to determine the application a document belongs to. The function AppFileType() returns the postfix for the file names of the documents created or processed by the application.

10.3.3 Class GraphicEditorDocument

The graphics editor shall be able to manage graphical documents , i.e. drawings. Such a drawing is a document in the sense of GINA++. Therefore, we have to define a subclass of GnDocument, which specializes some virtual functions to fit the new document class into the framework. This class also defines an application-specific callback handler and an embedded widget object for the palette.

```
class GraphicEditorDocument : public GnDocument {
    META_DEF_1(GraphicEditorDocument,GnDocument);
  public:
    GraphicEditorDocument();
  protected:
    // create widget tree
    virtual void CreateWindows(int new_width = 0, int new_height = 0);
  protected:
    // Resetting the document
    virtual void WipeOut();
  protected:
    // Document IO
    virtual XBoolean write_to_stream( ostream &output_stream );
    virtual XBoolean read_from_stream( istream &input_stream );
  protected:
    // Callback handlers for new commands
    void ClearAllObjects( caddr_t );
  protected:
    // Windows
    GnPixmapRadioGroup palette;
};
```

The function WipeOut() is called by the framework to reset the document (e.g. upon terminating the application). In our case, all graphical objects have to be deleted and the complete storage has to be released. The implentation can be found in appendix A.

The two functions write_to_stream() and read_from_stream() store a document and read it in again. For this purpose, the type, size and position of all graphical objects have to be stored. The implementation of these functions is straightforward, and can be found in appendix A.

The function ClearAllObjects() is devised as a callback for the "Clear All" menu entry (see below). Its purpose is to cause the deletion of all graphical objects. This operation is modelled by the command subclass ClearAll, so it can undone and redone. The function ClearAllObjects() has to create an object of this class, and must submit it to the framework. The framework then calls the function doit() on this command object.

```
void GraphicEditorDocument ::
ClearAllObjects(caddr_t)
{
    (new ClearAll( this ))->submit();
}
```

Creating the Widget Tree

The function CreateWindows() mainly defines the appearance and the interaction components of the application, i.e. the window(s) to be displayed upon starting the application, the subdivision of the windows, and what to display in them. This function has to be specialized by allmost all applications. First, it creates the main document shell, then all subwindows, and finally adds new menu entries to the menu bar.

```
void GraphicEditorDocument ::
CreateWindows(int new_width, int new_height)
{
    SetMainShell(new GnDocumentShell( this, new_width, new_height));
    GetMainShell()->create(APPLICATION->GetApplicationShell(), "docShell");

}
```

The framework passes the dimensions of the main shell to create_windows(). These dimensions are some default values in case a new document is created, or have been read from a document file if the document is an existing one.

The main shell contains a palette for selecting the drawing mode and a scrolled window will scroll bars containing a view for displaying the document's contents. These widgets are grouped together with a form widget.

```
    // Put everything in a form widget

    GnForm *form = new GnForm;
    form->create(this->GetMainShell(), "docForm");

    // Create and configure the palette

    palette.setR_orientation(XmVERTICAL);
    palette.setR_label_string("Select Drawing Mode");
    palette.setR_leftAttachment(XmATTACH_FORM);
    palette.setR_topAttachment(XmATTACH_FORM);
    palette.setR_shadowThickness(3);
    palette.SetButtonShadowThickness(3);
    palette.create(form, "palette");

    // Create and configure the scrolled window

    GnStaticScrolledWindow *scroller = new GnStaticScrolledWindow;
```

```
scroller->setR_leftAttachment(XmATTACH_WIDGET);
scroller->setR_leftWidget(palette);
scroller->setR_rightAttachment(XmATTACH_FORM);
scroller->setR_topAttachment(XmATTACH_FORM);
scroller->setR_bottomAttachment(XmATTACH_FORM);
scroller->create(form, "scrolledWindow");

// Create and configure the view

this->SetMainView(new GraphicEditorView(this));
this->GetMainView()->setR_width(1200);
this->GetMainView()->setR_height(1200);
this->GetMainView()->create(scroller, "mainView");
```

The code for adding items to the palette has been left out here and is presented later in the discussion of the class `DrawModeItem`. Both the form widget and the scrolled window are allocated on the heap and no references to them are needed later. They could have also been defined as embedded objects, in the same manner as the palette. The view is an instance of `GraphicEditorView`, which will be discussed below. A reference to it is assigned to the member variable `main_view`, which is inherited from the base class `GnDocument` and can be accessed via methods `GetMainView()` and `SetMainView()`.

Most of the code above deals with setting the constraint resources of the widgets. These resources determine the arrangement of the widgets relative to each other, and how the widgets behave when the main shell is resized either internally or from outside by the user. The arrangement is shown in figure 10.1. The behavior upon resizing is that the size of the palette remains constant, and that the scrolled window adapts its size to the new size of the main shell.

The dimensions of the view are fixed in the above code. In a more sophisticated application, they would be adjustable, and reasonable defaults would be chosen (e.g., DIN A4 size).

Adding Callbacks and Menu Entries

Next we register the function to be called when the state of the palette changes, i.e. when the user clicks on a palette icon and selects another graphical object type:

```
palette.add_selectionCallback(CALLBACK(GraphicEditorView,
                                       ChangeMode, GetMainView()));
```

When this callback is executed, the palette passes a reference to itself as the `call_data` argument to `ChangeMode()`. This reference can be used to query the current state of the palette.

Finally, we add a number of menu entries to the menu bar . The enty “Clear All” is used to delete all graphical objects created so far. The other entries determine the line width for graphical objects (see below).

```
GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "Clear All",
 CALLBACK(GraphicEditorDocument,
```

```
      ClearAllObjects, this));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "Thin Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,0));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "1 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,1));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "2 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,2));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "3 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,3));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "4 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,4));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "5 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,5));
```

Menu topics and entries are identified by their name. If a new name is encountered by the framework, then a new topic or entry is created automatically (i.e. "Draw"). Otherwise, the callback is added to the existing entry (see also 7).

10.3.4 Class `DrawModeItem` and `enum DrawMode`

The graphics editor may be either in *select mode* or in *draw mode*. In select mode, graphical objects may be selected, moved, and resized. In draw mode, new graphical objects may be drawn. The draw mode has six minor modes, each one corresponding to a different type of graphical object. Selecting the first icon (arrow) in the palette switches to select mode. Selecting any other icon switches to draw mode and selects one of the minor modes. Mode changes are propagated to the document with the function `GnView::ChangeMode()`, which has been attached as a callback to the palette (see above).

The enumeration type `DrawMode` describes the different modes of the graphics editor:

```
typedef enum DrawMode { eSelect, eRectangle, eSquare, eEllipse, eCircle,
                        eTriangle, eText };
```

The palette is realized with an widget of the class `GnPixmapRadioGroup`, which realizes a *1-of-many* interaction type. Items in this widget are presented by bitmap icons, and must be of either class `GnBitmapBitsPresentable` or `GnBitmapFilePresentable`. For our purpose, we use the class `GnBitmapBitsPresentable`, whose instances contain a two-dimensional bit field and its dimensions. These values usually come from an X11 bitmap file, which has been created with the X11 bitmap editor. Since we want to associate a mode with a bitmap icon, we derive the subclass `DrawModeItem` of `GnBitmapBitsPresentable`, which defines an additional field of type `DrawMode`:

```
class DrawModeItem : public GnBitmapBitsPresentable {
    META_DEF_1(DrawModeItem,GnBitmapBitsPresentable);
  public:
    DrawModeItem(char *p_bitmap_bits,
 unsigned int p_width, unsigned int p_height,
 DrawMode p_draw_mode);
    DrawMode GetDrawMode()
{ return draw_mode; }
  private:
    DrawMode draw_mode;
};

DrawModeItem::
DrawModeItem(char *p_bitmap_bits, unsigned int p_width, unsigned int p_height,
             DrawMode p_draw_mode)
: GnBitmapBitsPresentable(p_bitmap_bits, p_width, p_height)
{
    draw_mode = p_draw_mode;
}
```

Each item in the palette corresponds to an instance of `DrawModeItem`. The constructor of `DrawModeItem` passes the arguments describing the bitmap icon to the constructor of its base class and stores the associated mode on its member variable `draw_mode`.

The bitmap icons have been created with the X11 `bitmap` utility. They contain C code and can be included as header files:

```
#include "arrow"
#include "rectangle"
#include "square"
#include "ellipse"
#include "circle"
#include "triangle"
#include "glyph"
```

The code for adding items to the palette – which is part of the function `GnDocument::CreateWindows()` – then creates an item for each mode and adds it to the palette:

```
palette.add(new DrawModeItem(arrow_bits, arrow_width, arrow_height,
                             eSelect));
palette.add(new DrawModeItem(rectangle_bits, rectangle_width, rectangle_height,
                             eRectangle));
palette.add(new DrawModeItem(square_bits, square_width, square_height,
                             eSquare));
palette.add(new DrawModeItem(ellipse_bits, ellipse_width, ellipse_height,
                             eEllipse));
palette.add(new DrawModeItem(circle_bits, circle_width, circle_height,
                             eCircle));
```

```
palette.add(new DrawModeItem(triangle_bits, triangle_width, triangle_height,
                             eTriangle));
palette.add(new DrawModeItem(glyph_bits, glyph_width, glyph_height,
                             eText));
```

The code for accessing the currently selected item of the palette is shown below in the discussion of `GraphicEditorView`.

10.3.5 Class `GraphicEditorView`

A view displays (a part of) the contents of a document. The class `GnView` already has the capabilities for displaying, selecting, moving, and resizing graphical objects. Our task is to derive a subclass that adds capabilities for drawing new graphical objects interactively by direct-manipulation, and handles callbacks from the menu bar and from the palette.

```
class GraphicEditorView : public GnView {
    friend class GraphicEditorDocument;
  public:
    GraphicEditorView(GnDocument *p_document);
  public:
    // Accessors
    int GetDefaultLineWidth()
        { return default_line_width; }
  public:
    // Callback from the palette
    void ChangeMode(caddr_t call_data);

    // Callback from the menu bar
    void NewLineWidth(void *, caddr_t);
  protected:
    // Reaction to mouse input
    virtual void button_press( int code, int repetition, int x, int y );

    // Document IO (for view objects)
    void write_to_stream( ostream &ostr );
    void read_from_stream( istream &istr );
  protected:
    DrawMode drawing_mode;
    int      default_line_width;
};
```

The constructor initializes the default line width and puts the view into select mode:

```
GraphicEditorView ::
GraphicEditorView( GnDocument* p_document )
: GnView( p_document )
{
    drawing_mode       = eSelect;
    default_line_width = 1;
}
```

Callbacks from the palette

The function ChangeMode() has been added as a callback function to the palette, and is called whenever the user selects a new item.

```
void GraphicEditorView::
ChangeMode(caddr_t call_data)
{
    GnSingleSelection *sel = (GnSingleSelection *)(GnSelection *)call_data;


    DrawModeItem *item = DrawModeItem::TypeSafeDownCast(sel->SelectedItem());
    drawing_mode = item->GetDrawMode();
    PropagateButtonPress(drawing_mode == eSelect);
}
```

The call_data argument is converted into a pointer to GnSingleSelection, which is a base class of GnPixmapRadioGroup, the class of the palette. The new drawing mode is stored in the member variable drawing_mode. Additionally, the function PropagateButtonPress() is called. This functions determines whether button press events are propagated to the view objects installed in the view, or if they are mapped onto calls to the function button_press(). If the view is in select mode, events should be propagated to the view objects, so that selecting, moving, and resizing of view objects is handled automatically by the framework. In drawing mode, we want to handle button press events ourselves, so that we can create and submit mouse-down commands that create new graphical objects.

Reacting to Mouse Input

When the view is in drawing mode, all incoming button press events are mapped onto calls to the virtual function button_press(). We specialize this function in our class GraphicEditorView, so we can implement individual reactions to mouse clicks:

```
void GraphicEditorView ::
button_press( int /* code */, int /* repetition */, int x, int y )
{
    switch(drawing_mode) {
      case eRectangle:
(new RectangleDrawer( this, x, y ))->submit();
break;
      case eSquare:
(new SquareDrawer( this, x, y ))->submit();
break;
      case eEllipse:
(new EllipseDrawer( this, x, y ))->submit();
break;
      case eCircle:
(new CircleDrawer( this, x, y ))->submit();
break;
      case eSelect:
```

```
(new GnObjectSelector(this, x, y))->submit();
break;
      case eTriangle:
      case eText:
beep();
break;
    }
}
```

Depending on the current minor draw mode, we create a corresponding mouse-down command and submit it to the framework. These commands display a graphical feedback as long as the mouse button is pressed. Whenever the mouse moves while being pressed, the feedback adapts itself to the new mouse position. When the mouse button is released, the `doit()` function of the command is called (see also 9). This function creates the new graphical object and installs it in the view[1].

The functions for changing the line width and for reading and writing view objects are not discussed here. Their implementation can be found in appendix A.

10.3.6 Class `ClearAll`

The class `ClearAll` models a user command for deleting all graphical objects contained in a document. Such a command is created and submitted in the callback function `GraphicEditorDocument::ClearAllObject()`, which is attached to the "Clear All" menu entry:

```
void GraphicEditorDocument ::
ClearAllObjects(caddr_t )
{
    (new ClearAll( this ))->submit();
}
```

We first give the definition of `ClearAll`:

```
class ClearAll : public GnCommand {
  public:
    ClearAll( GnDocument * );
  protected:
    virtual char *name()
        { return("Clear All"); };
    virtual void doit();
    virtual void undoit();
    virtual void redoit();
  private:
    GnViewObjectList objects;
};
```

The virtual function `name()` returns the name of the command, which is displayed by the framework in the "Undo" and "Redo" menu entries and in the history scroller. The

[1] The commands for drawing triangles and for editing text have not been implemented yet

virtual function doit() implements the intended functionality of the command. Since we want the command to be undoable and redoable, we must keep enough information to be able to perform the inverse operation. In our case, this means that we must keep pointers to all graphical objects currently installed in the view. For this purpose, we define the private member objects, and collect all graphical objects into it in doit():

```
void ClearAll ::
doit()
{
    GnViewObjectIterator it(document->GetMainView()->get_view_objects());

    for( it.Start(); ! it.Off(); it.Forth() ) {
objects.Append(it.Item());
it.Item()->deinstall();
    }
}
```

The implementation of undoit() then iterates through the list of saved objects and (re)installs them into the view. This function is called by GINA++ either when the menu entry "Undo" is activated or when the command history is scrolled backward in the history scroller.

```
void ClearAll ::
undoit()
{
    for(GnViewObjectIterator it(objects); ! it.Off(); it.Forth() )
it.Item()->install(document->GetMainView(),
   it.Item()->get_x_pos(), it.Item()->get_y_pos() );
}
```

Finally, the redoit() function deinstalls all saved graphical objects again. This function is called by GINA++ either when the menu entry "Redo" is activated or when the command history is scrolled forward in the history scroller.

```
void ClearAll ::
redoit()
{
    for(GnViewObjectIterator it(objects); ! it.Off(); it.Forth() )
        it.Item()->deinstall();
}
```

10.3.7 Class ViewObjectDrawer

The class ViewObjectDrawer is an abstract base class for the mouse-down commands for drawing graphical objects. It defines the functionality common to these commands. The functionality that can not be expressed in the abstract base class must then be implemented in the concrete subclasses RectangleDrawer, SquareDrawer, EllipseDrawer and CircleDrawer.

```
class ViewObjectDrawer : public GnMouseDownCommand {
  public:
    ViewObjectDrawer( GnView *, int x, int y );
  protected:
    virtual void constrain_mouse( int &x, int &y );
    virtual void doit();
    virtual void undoit();
    virtual void redoit();
  protected: // New stuff
    virtual void create_view_object(int width, int height) = 0;
  protected:
    GnDirectManipulationObject *view_object;
};
```

This class specializes several inherited functions that integrate the new command class into the framework. There are two things this class can not implement: It can not deliver the name of the command by means of the virtaul function `name()`, and it can not create the new graphical object. All other things like constraining the mouse, doing, undoing, and redoing can be implemented. We introduce a new pure virtual function `create_view_object()`, which must be implemented in any subclass. This function is supposed to create a graphical object of the appropriate type, and to store it on the member variable `view_object`. We then call this function from `doit()`, and install the newly created view object into the view:

```
void ViewObjectDrawer ::
doit()
{
    constrain_mouse(last_x, last_y);

    int wid = ABS(last_x - start_x);
    int hei = ABS(last_y - start_y);

    create_view_object(wid, hei);
    view_object->
        setGC_line_width(((GraphicEditorView *)view)->GetDefaultLineWidth());
    view_object->install(view, start_x, start_y);
}
```

The dimensions and positions of the view object are determined with the inherited member variables `last_x`, `start_x`, `last_y`, and `start_y`.

The functions `undoit()` and `redoit()` install and deinstall the view object stored on `view_object` into the view, respectively.

```
void ViewObjectDrawer ::
undoit()
{
    view_object->deinstall();
}

void ViewObjectDrawer ::
```

```
redoit()
{
    view_object->install(view, start_x, start_y);
}
```

The virtual function `constrain_mouse()` is called by GINA++ whenever the mouse moves. Its purpose is to modify the actual mouse position when necessary. In our case, the mouse must not move to the left or above the position where it has been pressed, and there shall be at least a horizontal and vertical distance of 2 pixels from the original position:

```
void ViewObjectDrawer ::
constrain_mouse( int &x, int &y )
{
    x = MAX( x, start_x + 2 );
    y = MAX( y, start_y + 2 );
}
```

10.3.8 Classes `RectangleDrawer` and `SquareDrawer`

These classes realize mouse-down commands for drawing rectangles and circles. Since drawing a square is similar to drawing a rectangle, the class `SquareDrawer` inherits its behavior from `RectangleDrawer`.

```
class RectangleDrawer : public ViewObjectDrawer {
  public:
    RectangleDrawer( GnView *view, int x, int y );
  protected:
    virtual char *name() { return("Draw Rectangle"); };
  protected:
    virtual void create_view_object(int width, int height);
};

class SquareDrawer : public RectangleDrawer {
  public:
    SquareDrawer( GnView *view, int x, int y );
  protected:
    virtual char *name() { return("Draw Square"); };
  protected:
    virtual void constrain_mouse( int &x, int &y );
};
```

Both classes specialize the virtual function `name()`. The class `RectangleDrawer` specializes the function `create_view_object()`, which creates a rectangle object:

```
void RectangleDrawer::
create_view_object(int width, int height)
{
    view_object = new GnRectangleO( view->gc_values(), width, height );
}
```

The class GnRectangleO is part of GINA++ and models rectangular direct-manipulation objects that can be moved, resized, and selected.

The class SquareDrawer again specializes the function constrain_mouse:

```
void SquareDrawer::
constrain_mouse( int &x, int &y )
{
    int dist = MAX(2, MIN(ABS(x - start_x), ABS(y - start_y)));
    x = start_x + dist;
    y = start_y + dist;
}
```

The mouse position is constrained so that the horizontal and vertical distance from the position where the mouse has been initially pressed are equal. The function doit() inherited from ViewObjectDrawer then creates a rectangular object with equal width and height.

10.3.9 Classes EllipseDrawer and CircleDrawer

These classes are similar to RectangleDrawer and SquareDrawer. Their implementation can be found in appendix A.

Bibliography

[Bae91a] Andreas Baecker. "Designing Reusable Widget Classes with C++ and OSF/Motif". November 1991.

[Bae91b] Andreas Baecker. "Designing Reusable Widget Classes with C++ and OSF/Motif". Arbeitspapiere der GMD Nr. 590, November 1991.

[BBB+91] Andreas Baecker, Robert Babatz, Christian Beilken, Thomas Berlage, and Michael Spenke. *"CLM – A Language Binding for Common Lisp and OSF/Motif, Version 2.0"*. GMD, January 1991.

[Ber91a] Thomas Berlage. *"OSF/Motif, Concepts and Programming"*. Addison Wesley, Wokingham, England, 1991.

[Ber91b] Thomas Berlage. *"OSF/Motif und das X-Window System"*. Addison Wesley, Bonn, 1991.

[Col91] Jean-Christophe Collet. "C++ Products and Books List", May 1991.

[Fek91] Jean-Daniel Fekete. *"WWL, a Widget Wrapper Library for C++, Version 1.2"*. Laboratoire de Recherche en Informatique, Orsay Cedex, France, February 1991.

[GOP90] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *"Data Abstraction and Object-Oriented Programming in C++"*. John Wiley & Sons Ltd., Buffins Lane, Chichester, England, 1990.

[GR89] Adele Goldberg and David Robson. *"Smalltalk-80, The Language"*. Addison Wesley, Reading, MA, 1989.

[GSN87] James Gettys, Robert W. Scheifler, and Ron Newman. *"X Window System C Library and Protocol Reference"*. Digital Press, 1987.

[GSN89] James Gettys, Robert W. Scheifler, and Ron Newman. *"Xlib – C Language X Interface. X Version 11, Release 4"*, 1989.

[Jon89] Oliver Jones. *"Introduction to the X window system"*. Prentice-Hall, 1989.

[Kee88] Sonya E. Keene, editor. *"Object-Oriented Programming in Common Lisp"*. Addison Wesley, Reading, MA, 1988.

[KL89] Won Kim and Frederick H. Lockovsky, editors. *"Object-Oriented Concepts, Databases, and Applications"*. Addison Wesley, New York, 1989.

[KR78] Brian W. Kerninghan and Dennis M. Ritchie. *"The C Programming Language"*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Lip89] Stanley B. Lippmann. *"C++ Primer"*. Addison Wesley, Reading, MA, 1989.

[MAS89] Joel McCormack, Paul Asente, and Ralph R. Swick. *"X Toolkit Intrinsics – C Language Interface. X Version 11, Release 4"*, 1989.

[Mey88] Bertrand Meyer. *"Object-Oriented Software Construction"*. Prentice Hall, New York, 1988.

[Mey89] Bertrand Meyer. *"Eiffel: The Language, Version 2.2"*. Interactive Software Engineering, Inc (ICE), 1989.

[Mey92] Bertrand Meyer. *"Eiffel: The Language"*. Prentice Hall, Hertforshire, 1992.

[Ope90a] Open Software Foundation. *"Motif Programmer's Reference Manual"*, 1990. Version 1.1.

[Ope90b] Open Software Foundation. *"Motif Style Guide"*, 1990. Version 1.1.

[Ope90c] Open Software Foundation. *"Motif Toolkit Programmer's Guide"*, 1990. Version 1.1.

[Ope90d] Open Software Foundation. *"Motif User Interface Language Programmer's Guide"*, 1990. Version 1.1.

[Ope90e] Open Software Foundation. *"Motif Window Manager Programmer's Guide"*, 1990. Version 1.1.

[Ros88] S. H. Rosenthal. *"A Simple X Client, or, How hard can it really be to write 'Hello World'?"*, 1988.

[SB90a] Mike Spenke and Andreas Baecker. "GINA – An Application Framework based on a Language Binding for OSF/Motif and Common Lisp". In *Proceedings of the European X Window System Conference*, London, November 1990.

[SB90b] Mike Spenke and Christian Beilken. *"An Overview of GINA – the Generic Interactive Application"*. GMD, 1990.

[SBB+90a] Mike Spenke, Christian Beilken, Thomas Berlage, Andreas Baecker, and Robert Babatz. *"GINA User Manual, Version 1.0 for Common Lisp"*. GMD, November 1990.

[SBB+90b] Mike Spenke, Christian Beilken, Thomas Berlage, Andreas Baecker, and Robert Babatz. *"GINA Demo Applications, Version 1.0 for Common Lisp"*. GMD, November 1990.

[SBB+90c] Mike Spenke, Thomas Berlage, Andreas Baecker, Robert Babatz, and Christian Beilken. *"GINA Reference Manual, Version 1.0 for Common Lisp"*. GMD, November 1990.

[Sch86] K. J. Schmucker. *"Object Oriented Programming for the Macintosh"*. Hayden, Hasbrouck Heights, New Jersey, 1986.

[Smy90] David E. Smyth. *"Wcl – Widget Creation Library"*. Jet Propulsion Labs, California Institute Of Technology, Pasadena, CA, 1990.

[SRM90] K. Seetharaman, C. F. Rei, and B. R. Montague. *"A C++ Binding for OSF/Motif"*. University of Lowell, Lowell, MA, April 1990.

[Str87] Bjarne Stroustrup. *"The C++ Programming Language"*. Addison Wesley, Reading, MA, 1987.

[SW89] Ralph R. Swick and Terry Weissman. *"X Toolkit Athena Widgets – C Language Interface. X Version 11, Release 4"*, 1989.

[WGM89] A. Weinand, E. Gamma, and R. Marty. "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework". *Structured Programming*, 10(2), June 1989.

Index

Appendix A Source Code of the Graphics Editor

A.1 GraphicEditorApplication.h

```
#ifndef Greditapplication_h
#define Greditapplication_h

class GraphicEditorApplication : public GnApplication {
    META_DEF_1(GraphicEditorApplication,GnApplication);
  public:
    GraphicEditorApplication();
  protected:
    virtual GnDocument *CreateDocument();
  protected:
    virtual char *AppClass ()    { return "Gredit" ; }
    virtual char *AppSignature() { return "gredit" ; }
    virtual char *AppFileType()  { return ".gredit" ; }
};

#endif
```

A.2 GraphicEditorApplication.C

```
META_IMPL_1(GraphicEditorApplication,GnApplication,;);

GraphicEditorApplication::
GraphicEditorApplication()
{
}

GnDocument *GraphicEditorApplication ::
CreateDocument ()
{
    return new GraphicEditorDocument;
}
```

A.3 GraphicEditorDocument.h

```
#ifndef Greditdocument_h
#define Greditdocument_h

class GraphicEditorDocument : public GnDocument {
    META_DEF_1(GraphicEditorDocument,GnDocument);
  public:
    GraphicEditorDocument();
  protected:
    // create widget tree
    virtual void CreateWindows(int new_width = 0, int new_height = 0);
  protected:
    // Resetting the document
    virtual void WipeOut();
  protected:
    // Document IO
    virtual XBoolean write_to_stream( ostream &output_stream );
    virtual XBoolean read_from_stream( istream &input_stream );
  protected:
    // Callback handlers for new commands
    void ClearAllObjects( caddr_t );
  protected:
    // Windows
    GnPixmapRadioGroup palette;
};

#endif
```

A.4 GraphicEditorDocument.C

```
META_IMPL_1(GraphicEditorDocument,GnDocument,;);

GraphicEditorDocument::
GraphicEditorDocument()
{
}

void GraphicEditorDocument ::
CreateWindows(int new_width, int new_height)
{
    SetMainShell(new GnDocumentShell( this, new_width, new_height));
    GetMainShell()->create(APPLICATION->GetApplicationShell(), "docShell");

    // Put everything in a form widget

    GnForm *form = new GnForm;
    form->create(this->GetMainShell(), "docForm");
```

```
   // Add items to the palette

   palette.add(new DrawModeItem(arrow_bits, arrow_width, arrow_height,
eSelect));
   palette.add(new DrawModeItem(rectangle_bits, rectangle_width,
rectangle_height, eRectangle));
   palette.add(new DrawModeItem(square_bits, square_width, square_height,
eSquare));
   palette.add(new DrawModeItem(ellipse_bits, ellipse_width, ellipse_height,
eEllipse));
   palette.add(new DrawModeItem(circle_bits, circle_width, circle_height,
eCircle));
   palette.add(new DrawModeItem(triangle_bits, triangle_width,
triangle_height, eTriangle));
   palette.add(new DrawModeItem(glyph_bits, glyph_width, glyph_height,
eText));
   // Configuration of the palette

   palette.setR_orientation(XmVERTICAL);
   palette.setR_label_string("Select Drawing Mode");
   palette.setR_leftAttachment(XmATTACH_FORM);
   palette.setR_topAttachment(XmATTACH_FORM);
   palette.setR_shadowThickness(3);
   palette.SetButtonShadowThickness(3);
   palette.create(form, "palette");

   GnStaticScrolledWindow *scroller = new GnStaticScrolledWindow;
   scroller->setR_leftAttachment(XmATTACH_WIDGET);
   scroller->setR_leftWidget(palette);
   scroller->setR_rightAttachment(XmATTACH_FORM);
   scroller->setR_topAttachment(XmATTACH_FORM);
   scroller->setR_bottomAttachment(XmATTACH_FORM);
   scroller->create(form, "scrolledWindow");

   this->SetMainView(new GraphicEditorView(this));
   this->GetMainView()->setR_width(1200);
   this->GetMainView()->setR_height(1200);
   this->GetMainView()->create(scroller, "mainView");

   palette.add_selectionCallback(CALLBACK(GraphicEditorView,
  ChangeMode, GetMainView()));

   GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "Clear All",
    CALLBACK(GraphicEditorDocument,
     ClearAllObjects, this));
   GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "Thin Lines",
    CALLBACK1(GraphicEditorView,
      NewLineWidth,
      GetMainView(), int,0));
   GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "1 Pixel Lines",
    CALLBACK1(GraphicEditorView,
      NewLineWidth,
      GetMainView(), int,1));
```

```
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "2 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,2));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "3 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,3));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "4 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,4));
    GetMainShell()->GetMainMenu()->AddMenuCommand("Draw", "5 Pixel Lines",
     CALLBACK1(GraphicEditorView,
       NewLineWidth,
       GetMainView(), int,5));
}

void GraphicEditorDocument::
WipeOut()
{
    GnViewObjectList *list = GetMainView()->get_view_objects();
    GnViewObject     *obj;

    while( ! list->Empty() ) {
obj = list->First();
        obj->deinstall(False);
delete obj;
    }
}

void GraphicEditorDocument ::
ClearAllObjects(caddr_t )
{
    (new ClearAll( this ))->submit();
}

XBoolean GraphicEditorDocument ::
write_to_stream( ostream &ofs )
{
    ((GraphicEditorView*)GetMainView())->write_to_stream( ofs );
    return True;
}

XBoolean GraphicEditorDocument ::
read_from_stream ( istream &ifs )
{
    ((GraphicEditorView*)GetMainView())->read_from_stream( ifs );
    return True;
}
```

A.5 GraphicEditorView.h

```
// This may look like C code, but it is really -*- C++ -*-

//   Module      : GraphicEditorView.h   Version 1.3
//   LastSCCS    : 7/24/92  13:25:55
//   LastEdit    : "Thu Feb 20 17:46:34 1992"
//   Description :
//   Author      :
//   Copyright   : GMD Schloss Birlinghoven

#ifndef Greditview_h
#define Greditview_h

class GraphicEditorView : public GnView {
    friend class GraphicEditorDocument;
  public:
    GraphicEditorView(GnDocument *p_document);
  public:
    // Accessors
    int GetDefaultLineWidth()
{ return default_line_width; }
  public:
    // Callback from the palette
    void ChangeMode(caddr_t call_data);

    // Callback from the menu bar
    void NewLineWidth(void *, caddr_t);
  protected:
    // Reaction to mouse input
    virtual void button_press( int code, int repetition, int x, int y );

    // Document IO (for view objects)
    void write_to_stream ( ostream &ostr );
    void read_from_stream ( istream &istr );
  protected:
    DrawMode drawing_mode;
    int      default_line_width;
};

#endif
```

A.6 GraphicEditorView.C

```
GraphicEditorView ::
GraphicEditorView( GnDocument* p_document )
: GnView( p_document )
{
    drawing_mode       = eSelect;
    default_line_width = 1;
```

```
}

void GraphicEditorView::
ChangeMode(caddr_t call_data)
{
    GnSingleSelection *sel = (GnSingleSelection *)(GnSelection *)call_data;


    DrawModeItem *item = DrawModeItem::TypeSafeDownCast(sel->SelectedItem());
    drawing_mode = item->GetDrawMode();
    PropagateButtonPress(drawing_mode == eSelect);
}

void GraphicEditorView ::
button_press( int /* code */, int /* repetition */, int x, int y )
{
    switch(drawing_mode) {
      case eRectangle:
(new RectangleDrawer( this, x, y ))->submit();
break;
      case eSquare:
(new SquareDrawer( this, x, y ))->submit();
break;
      case eEllipse:
(new EllipseDrawer( this, x, y ))->submit();
break;
      case eCircle:
(new CircleDrawer( this, x, y ))->submit();
break;
      case eSelect:
(new GnObjectSelector(this, x, y))->submit();
break;
      case eTriangle:
      case eText:
beep();
break;
    }
}

void GraphicEditorView ::
NewLineWidth(void *client_data, caddr_t)
{
    int new_width = (int)client_data;
    XBoolean any_selected = False;
    GnViewObjectIterator iterator(view_objects);

    for ( iterator.Start(); ! iterator.Off(); iterator.Forth() ) {
if( iterator.Item()->is_selected() ) {
    iterator.Item()->setGC_line_width( new_width );
    any_selected = True;
}
    }
    if( ! any_selected ) {
```

```
default_line_width = new_width;
    }

    force_redraw();
}

void GraphicEditorView ::
write_to_stream( ostream &ofs )
{
    ofs << view_objects.Length() << endl;

    for(GnViewObjectIterator it(view_objects); ! it.Off(); it.Forth() )
it.Item()->write_to_stream( ofs );
}

void GraphicEditorView ::
read_from_stream ( istream &ifs )
{
    int nr_of_objects;
    GnViewObject *obj;

    ifs >> nr_of_objects;

    for ( int i = 0; i < nr_of_objects; ++i ) {
int obj_type;
ifs >> obj_type;
switch(obj_type) {
  case 1:
    obj = new GnRectangleO( gc_values(), 0, 0 );
    break;
  case 2:
    obj = new GnEllipseO( gc_values(), 0, 0 );
    break;
  default:
    Require(False); // Should be error_dialog ....
    break;
}

Ensure(obj_type == obj->IO_type() );

obj->read_from_stream( ifs );
obj->install( this );
    }
}
```

A.7 DrawModeItem.h

```
#ifndef Drawmodeitem_h
#define Drawmodeitem_h
```

```
typedef enum DrawMode { eSelect, eRectangle, eSquare, eEllipse, eCircle,
eTriangle, eText };

class DrawModeItem : public GnBitmapBitsPresentable {
    META_DEF_1(DrawModeItem,GnBitmapBitsPresentable);
  public:
    DrawModeItem(char *p_bitmap_bits,
 unsigned int p_width, unsigned int p_height,
 DrawMode p_draw_mode);
    DrawMode GetDrawMode()
{ return draw_mode; }
  private:
    DrawMode draw_mode;
};

#endif
```

A.8 DrawModeItem.C

```
META_IMPL_1(DrawModeItem,GnBitmapBitsPresentable,;);

DrawModeItem::
DrawModeItem(char *p_bitmap_bits, unsigned int p_width, unsigned int p_height,
     DrawMode p_draw_mode)
: GnBitmapBitsPresentable(p_bitmap_bits, p_width, p_height)
{
    draw_mode = p_draw_mode;
}
```

A.9 ClearAll.h

```
#ifndef Clearall_h
#define Clearall_h

class ClearAll : public GnCommand {
  public:
    ClearAll( GnDocument * );
  protected:
    virtual char *name()
{ return("Clear All"); };
    virtual void doit();
    virtual void undoit();
    virtual void redoit();
  private:
    GnViewObjectList objects;
};
```

```
#endif
```

A.10 ClearAll.C

```
ClearAll ::
ClearAll( GnDocument *doc )
: GnCommand( doc )
{
}

void ClearAll ::
doit()
{
    GnViewObjectIterator it(document->GetMainView()->get_view_objects());

    for( it.Start(); ! it.Off(); it.Forth() ) {
objects.Append(it.Item());
it.Item()->deinstall();
    }
}

void ClearAll ::
undoit()
{
    for(GnViewObjectIterator it(objects); ! it.Off(); it.Forth() )
it.Item()->install(document->GetMainView(),
   it.Item()->get_x_pos(), it.Item()->get_y_pos() );
}

void ClearAll ::
redoit()
{
    for(GnViewObjectIterator it(objects); ! it.Off(); it.Forth() )
        it.Item()->deinstall();
}
```

A.11 ViewObjectDrawer.h

```
#ifndef Viewobjectdrawer_h
#define Viewobjectdrawer_h

class ViewObjectDrawer : public GnMouseDownCommand {
  public:
    ViewObjectDrawer( GnView *, int x, int y );
  protected:
    virtual void constrain_mouse( int &x, int &y );
    virtual void doit();
    virtual void undoit();
```

```
    virtual void redoit();
  protected: // New stuff
    virtual void create_view_object(int width, int height) = 0;
  protected:
    GnDirectManipulationObject *view_object;
};

#endif
```

A.12 ViewObjectDrawer.C

```
ViewObjectDrawer::
ViewObjectDrawer( GnView *view, int x, int y )
: GnMouseDownCommand( view->get_document(), view, x, y )
{
    view_object = 0;
}

void ViewObjectDrawer ::
constrain_mouse( int &x, int &y )
{
    x = MAX( x, start_x + 2 );
    y = MAX( y, start_y + 2 );
}

void ViewObjectDrawer ::
doit()
{
    constrain_mouse(last_x, last_y);

    int wid = ABS(last_x - start_x);
    int hei = ABS(last_y - start_y);

    create_view_object(wid, hei);
    view_object->
setGC_line_width(((GraphicEditorView *)view)->GetDefaultLineWidth());
    view_object->install(view, start_x, start_y);
}

void ViewObjectDrawer ::
undoit()
{
    view_object->deinstall();
}

void ViewObjectDrawer ::
redoit()
{
    view_object->install(view, start_x, start_y);
}
```

A.13 RectangleDrawer.h

```
#ifndef Rectangledrawer_h
#define Rectangledrawer_h

class RectangleDrawer : public ViewObjectDrawer {
  public:
    RectangleDrawer( GnView *view, int x, int y );
  protected:
    virtual char *name() { return("Draw Rectangle"); };
  protected:
    virtual void create_view_object(int width, int height);
};

#endif
```

A.14 RectangleDrawer.C

```
RectangleDrawer::
RectangleDrawer( GnView *view, int x, int y )
: ViewObjectDrawer(view, x, y)
{
}

void RectangleDrawer::
create_view_object(int width, int height)
{
    view_object = new GnRectangleO( view->gc_values(), width, height );
}
```

A.15 SquareDrawer.h

```
#ifndef Squaredrawer_h
#define Squaredrawer_h

class SquareDrawer : public RectangleDrawer {
  public:
    SquareDrawer( GnView *view, int x, int y );
  protected:
    virtual char *name() { return("Draw Square"); };
  protected:
    virtual void constrain_mouse( int &x, int &y );
};
```

```
#endif
```

A.16 SquareDrawer.C

```
SquareDrawer::
SquareDrawer( GnView *view, int x, int y )
: RectangleDrawer(view, x, y)
{
}

void SquareDrawer::
constrain_mouse( int &x, int &y )
{
    int dist = MAX(2, MIN(ABS(x - start_x), ABS(y - start_y)));
    x = start_x + dist;
    y = start_y + dist;
}
```

A.17 EllipseDrawer.h

```
#ifndef Ellipsedrawer_h
#define Ellipsedrawer_h

class EllipseDrawer : public ViewObjectDrawer {
  public:
    EllipseDrawer( GnView *view, int x, int y );
    virtual char *name() { return("Draw Ellipse"); };
  protected:
    virtual void draw_feedback(int x, int y, XBoolean clear );
    virtual void create_view_object(int width, int height);
};

#endif
```

A.18 EllipseDrawer.C

```
EllipseDrawer ::
EllipseDrawer( GnView *view, int x, int y )
: ViewObjectDrawer(view, x, y )
{
}

void EllipseDrawer::
create_view_object(int width, int height)
```

```
{
    view_object = new GnEllipseO( view->gc_values(), width, height );
}

void EllipseDrawer ::
draw_feedback(int x, int y, XBoolean clear )
{
    Gina_Debug( "EllipseDrawer :: draw_feedback " );
    Gina_Debug( MIN(start_x, x) ); Gina_Debug( " " );
    Gina_Debug( MIN(start_y, y) ); Gina_Debug( " " );
    Gina_Debug( ABS(x - start_x) ); Gina_Debug( " " );
    Gina_Debug( ABS(y - start_y) ); Gina_Debug( " " );
    Gina_Debug_NL( clear ? "clear" : "paint" );

    XDrawArc(view->get_display(), view->get_x_window(), view->get_feedback_gc(),
     MIN(start_x, x), MIN(start_y, y),
     ABS(x - start_x), ABS(y - start_y), 0, 64*360);
}
```

A.19 CircleDrawer.h

```
#ifndef Circledrawer_h
#define Circledrawer_h

class CircleDrawer : public EllipseDrawer {
  public:
    CircleDrawer( GnView *view, int x, int y );
  protected:
    virtual void constrain_mouse( int &x, int &y );
};

#endif
```

A.20 CircleDrawer.C

```
CircleDrawer::
CircleDrawer( GnView *view, int x, int y )
: EllipseDrawer(view, x, y)
{
}

void CircleDrawer::
constrain_mouse( int &x, int &y )
{
    int dist = MAX(2, MIN(ABS(x - start_x), ABS(y - start_y)));
    x = start_x + dist;
    y = start_y + dist;
```

```
}
```

A.21 The Program Body (gredit.C)

```
#include <Gina/Gina.h>
#undef Boolean

#include "arrow"
#include "rectangle"
#include "square"
#include "ellipse"
#include "circle"
#include "triangle"
#include "glyph"

#include "GraphicEditorApplication.h"
#include "GraphicEditorDocument.h"
#include "DrawModeItem.h"
#include "GraphicEditorView.h"
#include "ViewObjectDrawer.h"
#include "RectangleDrawer.h"
#include "SquareDrawer.h"
#include "EllipseDrawer.h"
#include "CircleDrawer.h"
#include "ClearAll.h"

#include "GraphicEditorApplication.C"
#include "GraphicEditorDocument.C"
#include "DrawModeItem.C"
#include "GraphicEditorView.C"
#include "ViewObjectDrawer.C"
#include "RectangleDrawer.C"
#include "SquareDrawer.C"
#include "EllipseDrawer.C"
#include "CircleDrawer.C"
#include "ClearAll.C"

int main (unsigned int argc, char ** argv)
{
    GraphicEditorApplication gredit;

    gredit.AddFallbackResource("*docShell.geometry: 600x400");

    gredit.Run( argc, argv );
}
```