
Class Libraries & Frameworks

Wolfgang Pree

C. Doppler Laboratory for Software Engineering
University of Linz

How to create flexible software systems

Data and program dualism

Architecture flexibility in
class libraries & frameworks

Frameworks

Single components versus frameworks

Black-box vs. white-box frameworks

Abstract classes & abstract coupling

Case study: GUI frameworks

How to create flexible software

How to create flexible software

Adele Goldberg in her OOPSLA'95 keynote speech:

Most programmers are just accomplishing their programming tasks by **building rigid systems that solve specific problems.**

The challenge is to teach **how to create maintainable systems which offer adequate flexibility for evolutionary changes.**

OO technology represents just one, albeit important means among others to achieve flexibility!

How to create flexible software

Flexibility based on data and program dualism:

Both programs and data are represented digitally as bit streams in computer memory

=> they can be considered as being equal at the bit stream level of abstraction

This duality of programs and data means that **a program can process another one as input data**. The output data could be a transformed program.

Of course, such a transformation could also be applied to the transforming program itself.

How to create flexible software

Application of this dualism in the early days of programming:

Some machine and assembly language programs became **self-modifying** ones—**nightmares for those who tried to understand what is going on.**

Fortunately, **high-level languages** introduced **more elegant ways of keeping software flexible.**

The concept of

- variables

together with basic control structures

- selection and
- loop statements

offer the required flexibility to express any algorithm.

How to create flexible software

Based on these language constructs, **modifying programs by modifying their input data** has become the most widely used conventional way of adapting software systems:

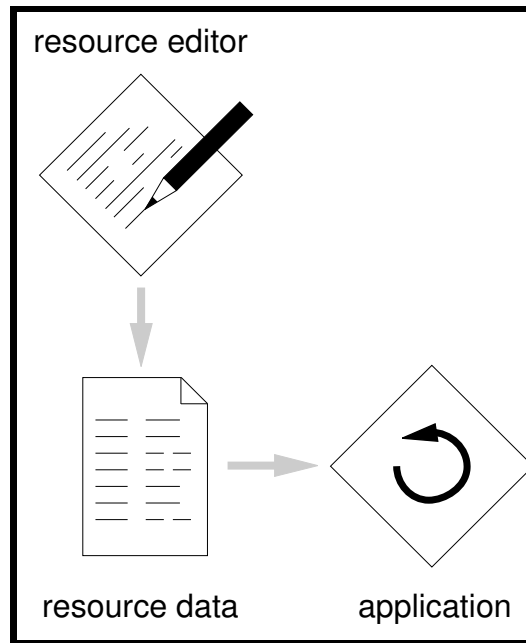
The **straight-forward idea** behind this adaptation strategy is that the values of variables and thus a software system's behavior is modified by editing data that are stored separately.

The program acts quasi as interpreter of the data stored in separate files.

=> **Adaptations require no recompilation and linking.**

How to create flexible software

Resources as sample application of this adaptation strategy:



The **structure and complexity of these data** may range from

- simple lists of values
e.g., for feeding a calculation component in an application
- highly structured, complex data
e.g., for describing the user interface layout

How to create flexible software

The resource concept **imposes no restrictions** on the type of application and **how the application is designed and implemented**.

In other words, **resources work equally well with conventional and object-oriented programs**.

Example: OO hotel reservation system

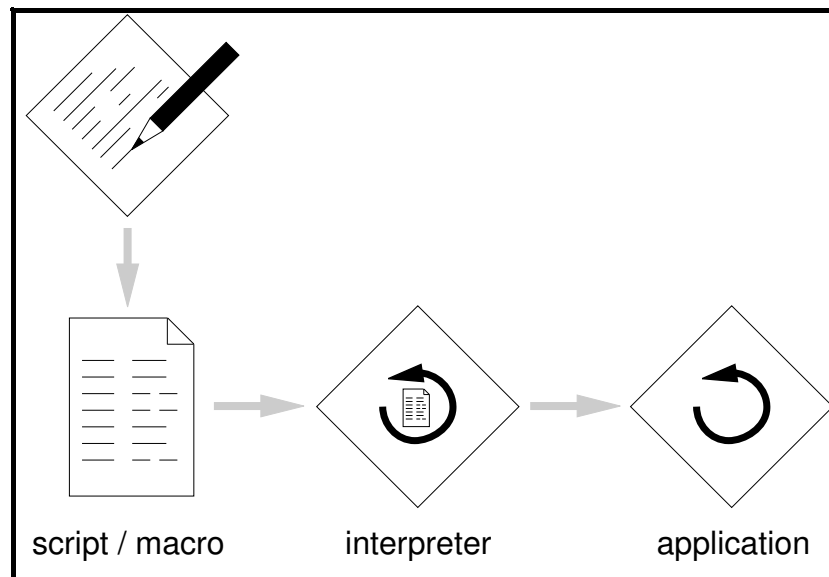
A resource file might specify **which rate calculation class should be instantiated** upon application startup.

Thus the rate calculation behavior of the reservation system can be **modified by editing this aspect of its configuration resource**.

How to create flexible software

Scripting languages, macros:

Though scripts and macros are conceptually **just another specific type of resource**, the overall systems differ in that they **process resources typically by an explicit interpreter**, separated from a particular application:



How to create flexible software

In this case, resource data already have a program flavor. The **interpreter processes script or macro resources and initiates actions** based on the commands found in these resources.

A scripting language describes the rules of how to write valid scripts. For instance, a(n attributed) grammar lends itself for specifying a scripting language in Extended Backus-Naur Form (EBNF) notation.

How to create flexible software

Sample scripting language systems:

- TCL

TCL (Tool Command Language; Sun microsystems) lets programmers (or even end users) glue together various applications or application components.

A TCL script consists of **TCL command strings**. Based on the information provided in such strings, the **TCL interpreter invokes the corresponding procedure of the application targeted in a particular command line** of the script.

In doing that, the **TCL interpreter quasi links together different application components**.

- spreadsheet macros
- . . .

How to create flexible software

Architecture flexibility through advanced programming language concepts:

programming language constructs with an abstraction level higher than variables and control flow statements:

- routines (that is, procedures and functions), and
- classes

In general, **more flexibility implies more programming effort by the programmers who reuse the software components.**

The ordering below reflects **decreasing levels of flexibility and thus increasing ease of reuse.**

How to create flexible software

Routine & ordinary class libraries:

Structured, function-oriented programming languages provide **routines as building blocks**.

Object-oriented languages offer **classes**.

A lot of domain-specific routine and class libraries have been developed, e.g.:

- mathematical libraries
- GUI libraries
- networking libraries
- database libraries.

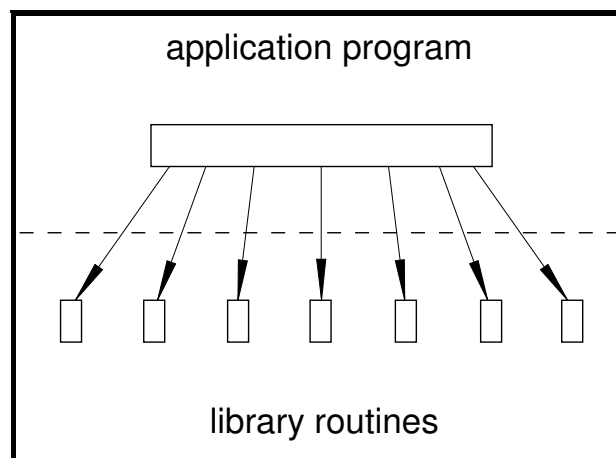
In the following we discern between

- top-down routine libraries
- class libraries and
- routine libraries with a callback style.

How to create flexible software

Top-down routine libraries:

- Do not provide any application structure
=> application programmers have to define the overall system architecture



- Might be compared to nuts and bolts in physical systems
- Top-down decomposition does not take into account the structuring of data.
=> stack of dominoes effect: changes to data structures and/or routines result in a multitude of changes

How to create flexible software

Ordinary class libraries:

- They don't provide any application structure, similar to top-down routine libraries.

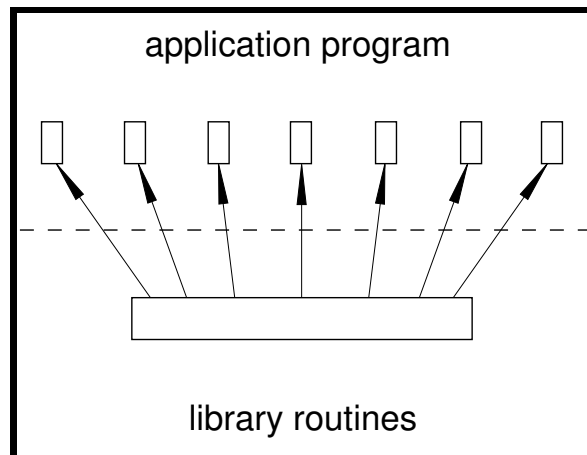
The difference is that classes grouping related routines **represent a higher level of abstraction than routines.**

Furthermore, **inheritance allows changes** of classes **without touching their source code.**

How to create flexible software

Routine libraries with a callback style:

- invert the control flow



Core parts of the application architecture reside in the library routines, not in the application.

Flexibility of the application architecture is achieved as follows: The **library routines *call out* to various routines which the application has previously registered with the library routines.**

This means that the application programmer passes specific routines as parameter values to library routines, which is, for example, possible in the language C.

How to create flexible software

Example: event loop in GUI applications

The core activity of an application with a GUI is to **constantly look for input events**, that is, mouse actions, and keystrokes, that occur in any order.

This application structure is called an **event loop**.

The whole **event loop** including necessary window system initializations **could go into a library routine Run()** if the application-specific parts could be added later on:

At a certain point it calls the routine **UserAction()** which is provided as parameter:

```
void Run(void (*UserAction)()) {  
    while (TRUE) {  
        /* ... */  
        UserAction();  
        /* ... */  
    }  
}
```

How to create flexible software

A programmer uses `Run()` by providing a routine that accomplishes the application-specific processing of events:

```
void MyAction() {  
    /* ... */  
}
```

=> programmer calls `Run(MyAction);`

Routine libraries with a callback style **restrict the programmer's flexibility**. Functionality can only be added where the developers of library routines earmark that possibility.

Actually, the flavor of **working with such libraries comes very close to modifying object-oriented frameworks through inheritance, that is, white-box frameworks** (see below).

Frameworks

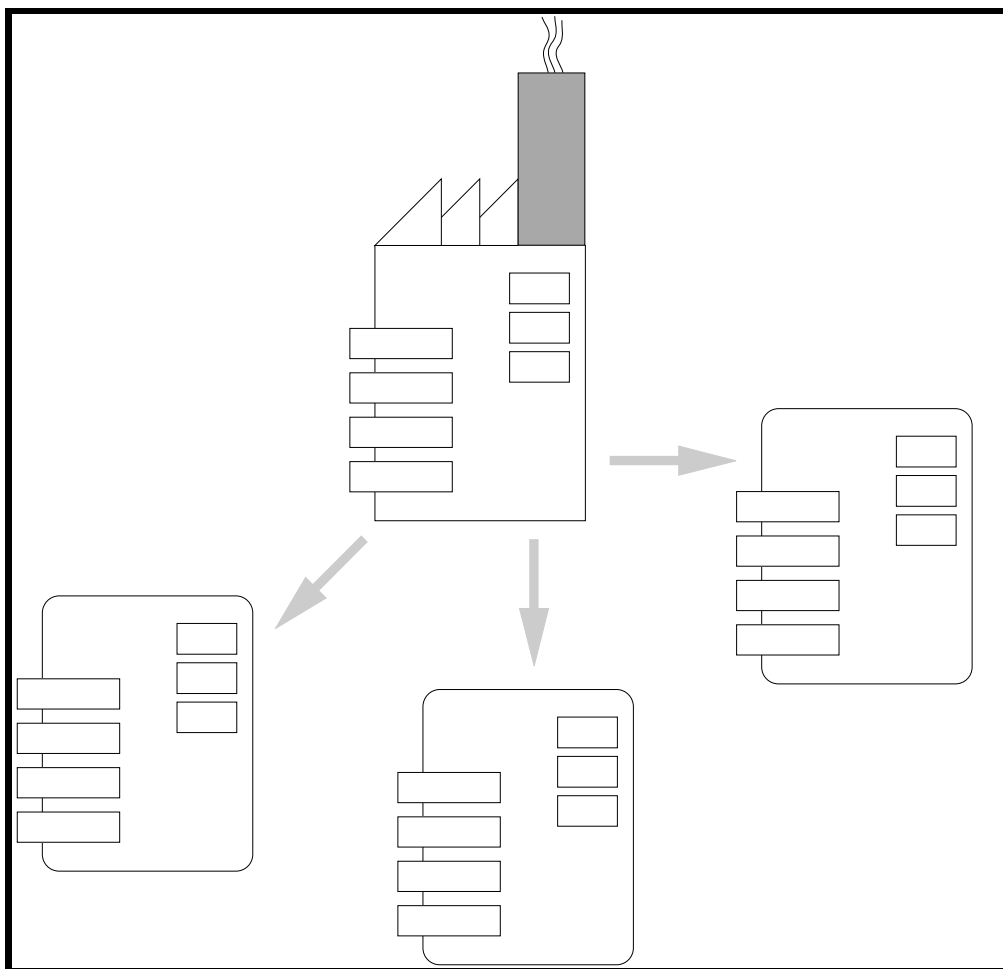
Single component reuse

Single component reuse (class libraries) versus frameworks:

OO languages are applied like slightly enhanced module-oriented languages in order to produce single components:

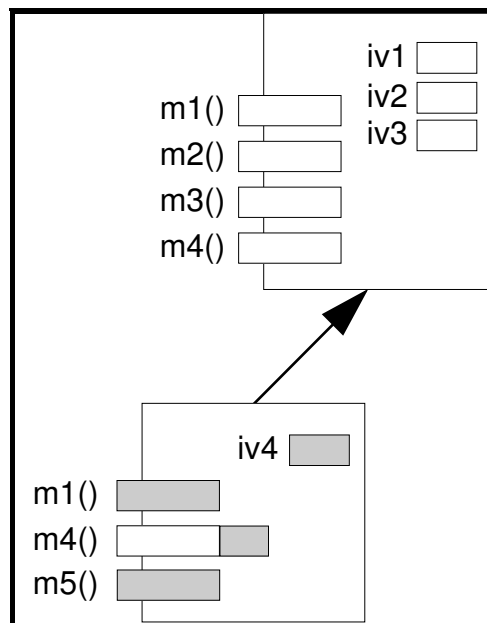
- Classes represent instantiable modules:

Single component reuse

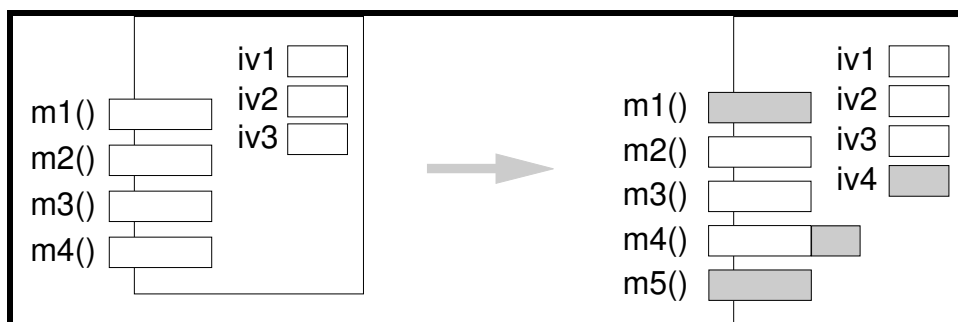


Single component reuse

- Inheritance allows module adaptations without having to edit source code and giving up compatibility:



flat class view:



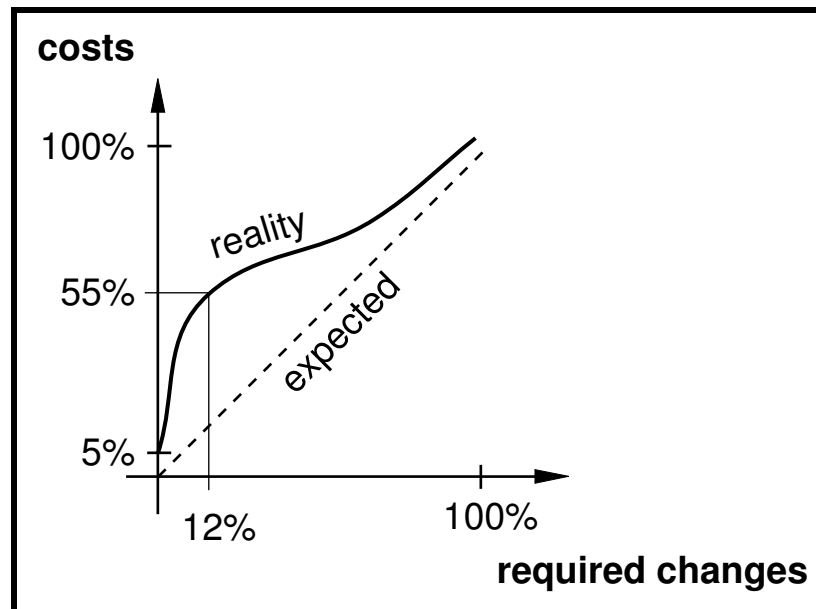
Single component reuse

Experience:

only a **marginal improvement** of reusability is achievable (compared to conventional solutions)

- only simple components are reusable as black boxes (data structures, GUI items, etc.)
- complex ones are too project-specific

Costs of single component reuse (from B. Boehm, 1994):



Frameworks

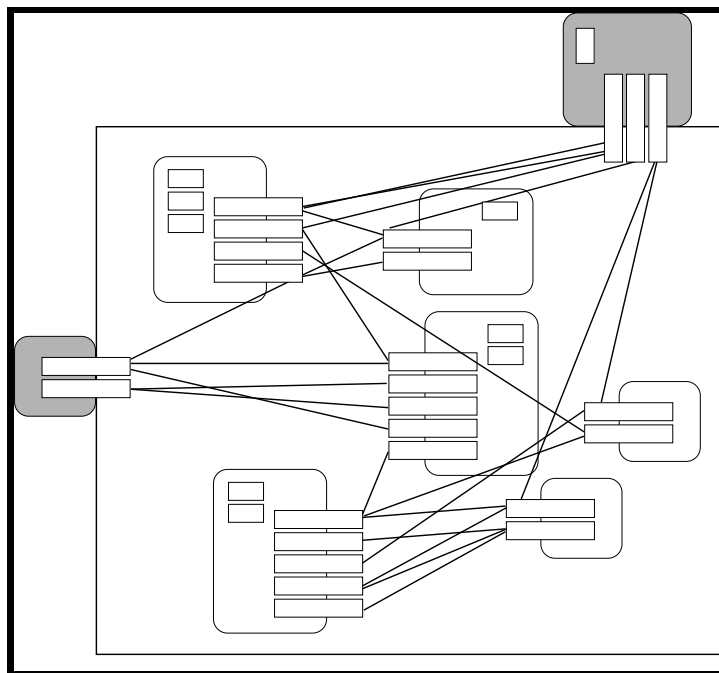
Only

reusable software architectures \Leftrightarrow

frameworks \Leftrightarrow

single components + interaction

allow the exploitation of the full potential of OO for a wide range of application domains.



Frameworks

Frameworks...

- + allow reuse of **architecture design + code**
 - => significantly reduced development & maintenance costs
 - => standardized application structure

- + can be produced for almost any commercial and technical application domain

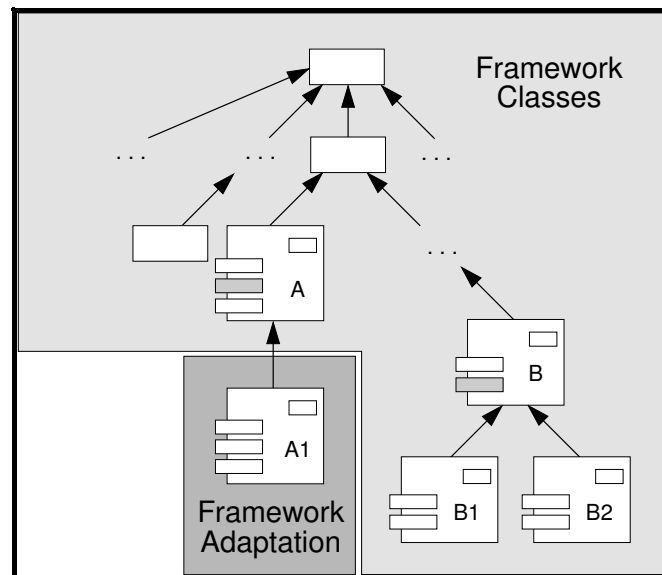
- **enormous development effort** (requires detailed domain knowledge)
 - => long-term investment
 - => pay-off if similar applications are developed for a domain

- at odds with current project culture

White-box frameworks

White-box frameworks consist of **several incomplete classes**, that is, classes that contain methods without meaningful default implementations.

Class A in the following sample framework class hierarchy illustrates this characteristic of a white-box framework:



The abstract method of class A that has to be overridden in a subclass is drawn in gray color.

White-box frameworks

In order to **modify the behavior** of white-box frameworks, **programmers apply inheritance** to override methods in subclasses of framework classes.

Such method (re)definitions are **analogous to providing specific functions in routine libraries with a callback style of programming**: The application architecture resides in the framework. Programmers adapt the framework by overriding the hook methods called out from other methods in the framework.

The necessity to override methods implies that **programmers have to understand the framework's design and implementation**, at least to a certain degree of detail.

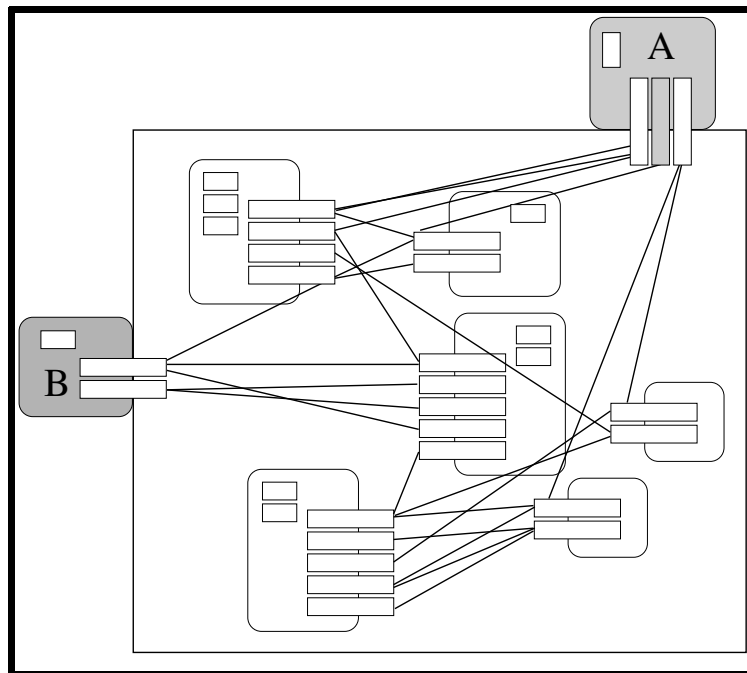
The **principal difference between routine libraries with a callback style and white-box frameworks** is analogous to the difference between top-down routine libraries and class libraries, namely that **the class construct allows a higher level structuring** of the software system.

Black-box frameworks

Black-box frameworks offer **ready-made components for adaptations**. Modifications are accomplished by composition, not by programming.

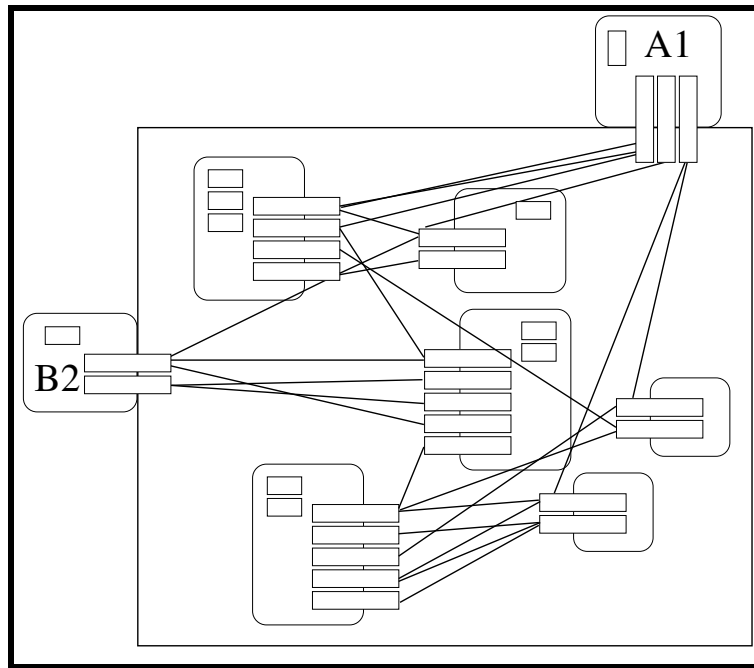
In the sample framework class hierarchy shown above, class **B** has already two subclasses **B1** and **B2** that provide default implementations of **B**'s abstract method.

Supposed that the framework components interact as follows:



Black-box frameworks

=> a programmer adapts this framework, for example, by instantiating classes **A1** and **B2** and plugging in the corresponding objects:



In case of class **B** the framework provides ready-to-use subclasses, in case of class **A** the programmer has to subclass **A** first.

Black-box frameworks

Available frameworks are **neither pure white-box nor pure black-box frameworks**.

If the framework is heavily reused, numerous specializations will suggest which black-box defaults could be offered instead of just providing a white-box interface.

So frameworks will evolve more and more into black-box frameworks when they mature.

Frameworks & Componentware

Component standards such as the object models **COM** and **(D)SOM** make different languages and compilers interoperable.

OLE and **OpenDoc** rest on top of the particular underlying object model and **manage resources** (keyboard, screen) between interoperating components.

Nevertheless, **Plug & Play** of such components **does not come for free**.

Frameworks form the underlying technology of component systems whose behavior is modified and/or extended by composition.

Core construction principle

Abstract classes & abstract coupling:

Classes that define common behavior usually do not represent instantiable classes but abstractions of them. They are called **abstract classes**.

It does not make sense to generate instances of abstract classes since some methods are abstract and have empty/dummy implementations. The general idea behind abstract classes is clear and straightforward:

- Properties (that is, instance variables and methods) of similar classes are defined in a common superclass.
- Some methods of the resulting abstract class can be implemented, while only dummy or preliminary implementations can be provided for others, which are termed **abstract methods**.

Core construction principle

Though abstract methods cannot be implemented, their names and parameters are specified since descendants cannot change the method interface. So an abstract class creates a **standard class interface** for all descendants.

Instances of all descendants of an abstract class will understand at least all messages that are defined in the abstract class.

Core construction principle

The **implication of abstract classes is that other software components based on them can be implemented.**

These components rely on the protocol supported by the abstract classes, that is they are abstractly coupled with the abstract classes. Most important, **these components interact properly, that is without change and recompilation, with instances of all future extensions of the abstract classes.**

Core construction principle

The key problem is to find useful abstractions so that software components can be implemented without knowing the specific details of concrete objects.

The **exercises** (collection classes, change propagation, reservation system, mailing system, simulation system) **illustrate** among other aspects **the central role of abstract classes in connection with abstract coupling in the realm of framework design.**

GUI frameworks

GUI frameworks

State-of-the-art GUI frameworks support the construction of applications as originally known from the Lisa and Apple Macintosh computers:

- **applications** allow the manipulation of various information structures (text, trees, spreadsheet cells, etc.)
- **documents** are created by means of such applications; documents are rendered in **windows**; the **inner part of a window views the data** adhering to the WYSIWYG principle and is thus called a **view**.
- **commands** are entered in post-order: the user first selects an object and then chooses from a menu a command message to be sent to that object.

Commands should be undoable.

GUI frameworks

Some commands can be performed directly by manipulating an object (→ **direct manipulation**).
Examples: deleting an iconized document by moving it into a trash can, drag&drop operations in text editors.

Important design principle: **avoid modes!**

GUI frameworks

Users trigger events in almost any order by doing mouse clicks and key strokes. Thus the **overall structure of a GUI application** is the following "endless" **event loop** which ends when the application quits:

- wait for events coming from the mouse or keyboard
- analyze these events and relate them to actions; the actions might cause a change in data structures and their rendering on the display
- wait for the next input events

GUIs can be implemented in a conventional way or by means of object-oriented frameworks.

GUI frameworks

Pros and cons of conventional, call-back routine libraries:

- + easy to handle if hook routines are available
- + sufficient for dialogs where GUI dialog items are used instead of character-oriented dialog items
- applications relying on direct manipulation are quite difficult to implement
- changes other than the foreseen ones are impossible; overall such libraries turn out to be inflexible
- complexity

Examples: Xt, SDK, XVT, Motif- and Openlook Toolkits

GUI frameworks

Pros and cons of GUI frameworks:

- ± programming flavor is similar to call back libraries
- + can be significantly better parameterized
 - => increased flexibility
 - => more complex control flow can be factored out into library components
 - => direct manipulation style can be supported
- complexity

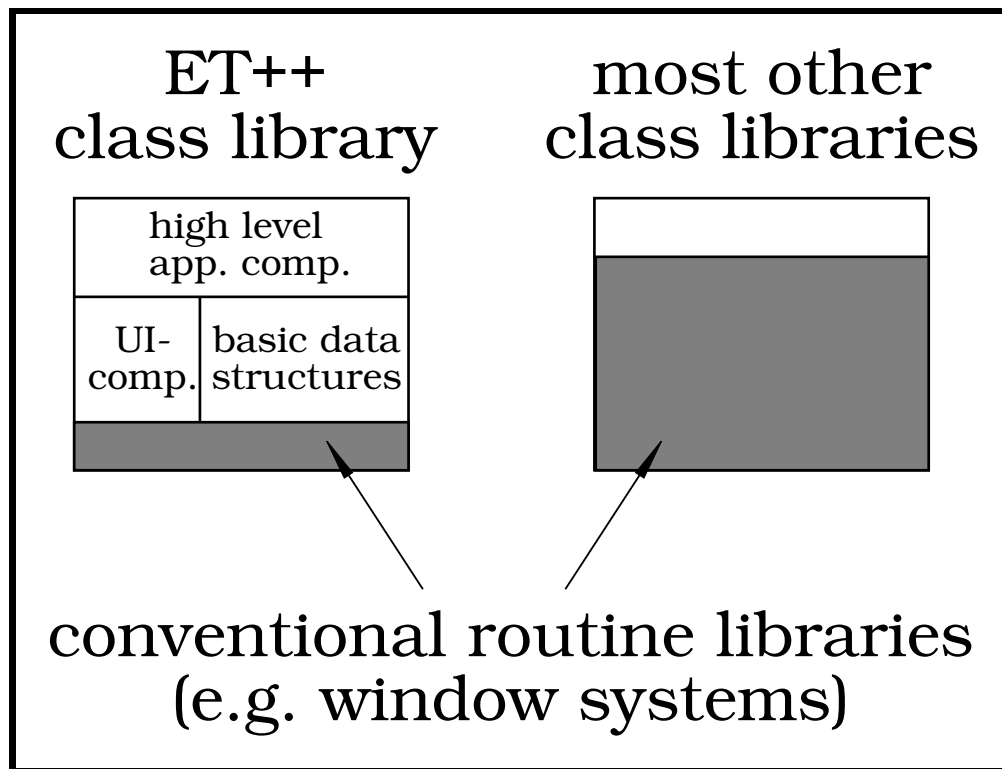
Examples: ET++, Commonpoint (Taligent), Smalltalk frameworks, MacApp, AppKit (NeXT), Interviews, Microsoft's Foundation Classes (MFC), Oberon/F, Java UI toolkit (AWT).

GUI frameworks

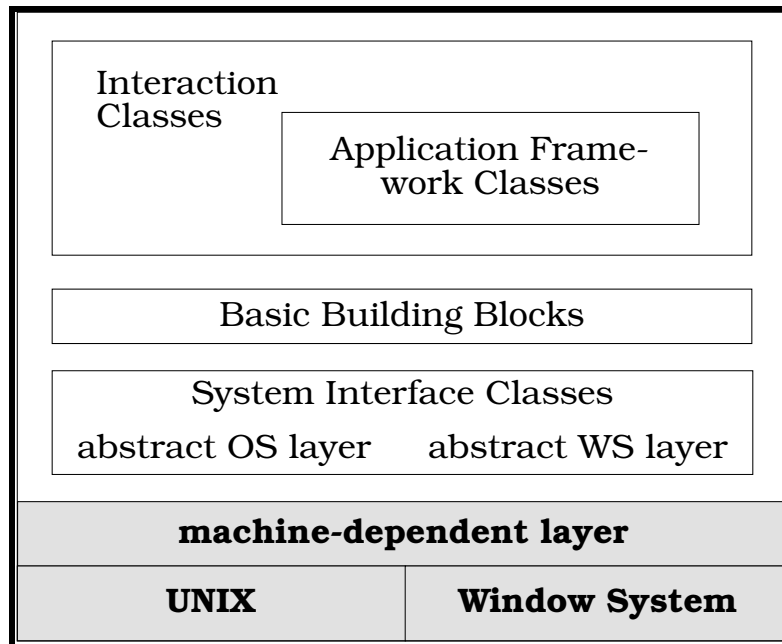
Typical tasks of what a full-fledged GUI framework such as ET++ automatically takes care of:

- management of an arbitrary number of windows with their models and the visual representation of these models (abstract classes **Document** and **View**)
- moving, resizing, opening, closing and stacking of windows
- file and dialog management for loading and storing **Document** objects
- scrolling the window contents (auto-scrolling, real-time-scrolling: realized by **Scroller** and **Splitter** based on the abstract class **VObject**)
- flicker-free screen update using double buffering
- device-independent hardcopy output (e.g., in **PostScript**)
- undoable commands (abstract class **Command**)

GUI frameworks



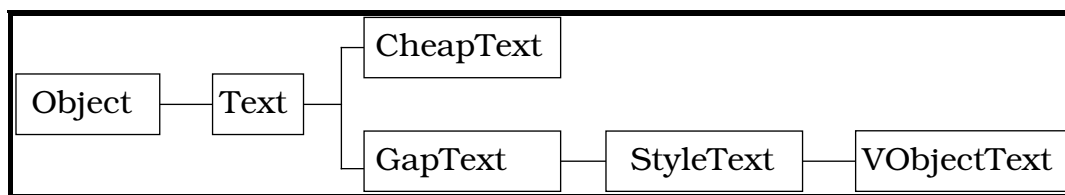
GUI frameworks



GUI frameworks

Basic building blocks:

- class `Object`
- collection classes
- text classes



`CheapText`: small text pieces, only one style

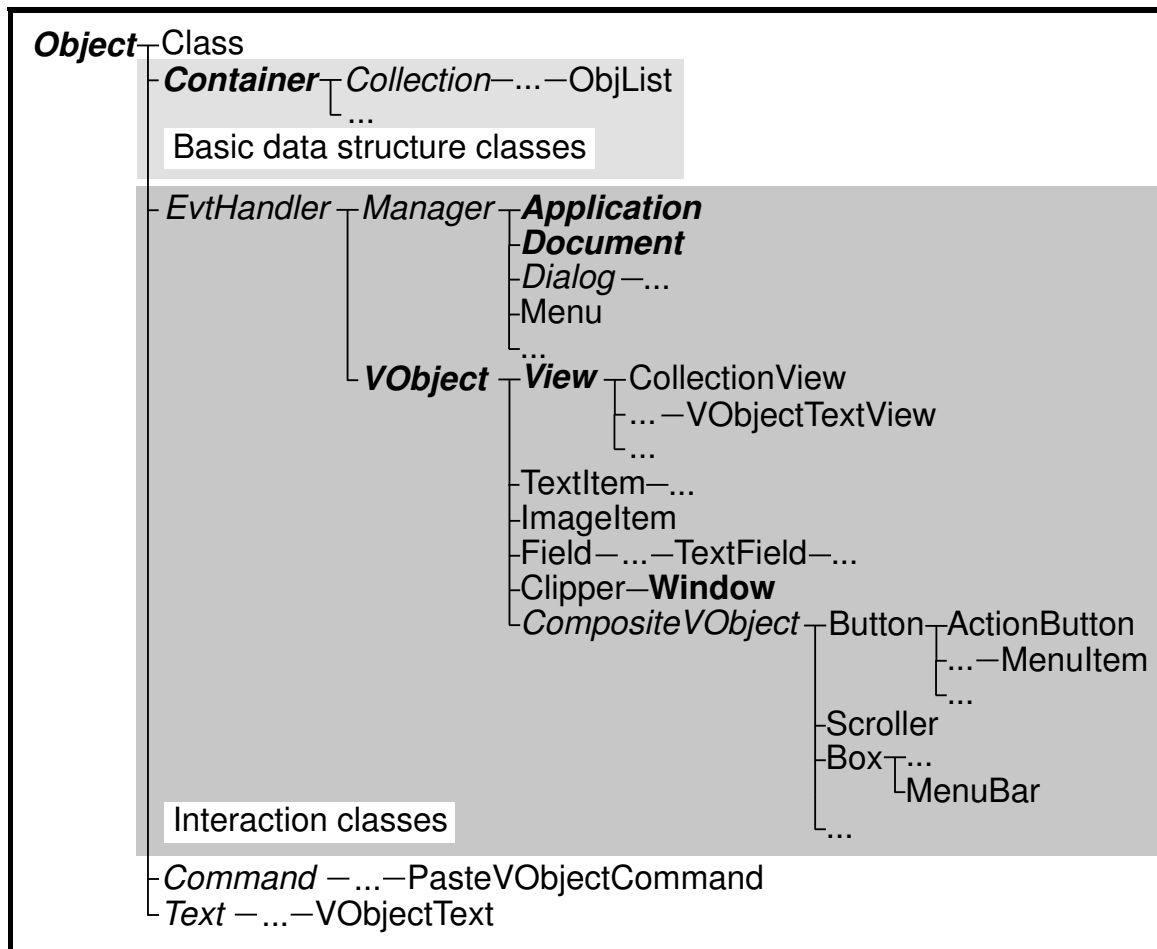
`GapText`: large texts, only one style

`StyleText`: allows any number of styles

`VObjectText`: allows embedded graphic objects

GUI frameworks

Hierarchy of important ET++ classes:



GUI frameworks

Graphic Building Blocks / Interaction Classes:

- class VObject

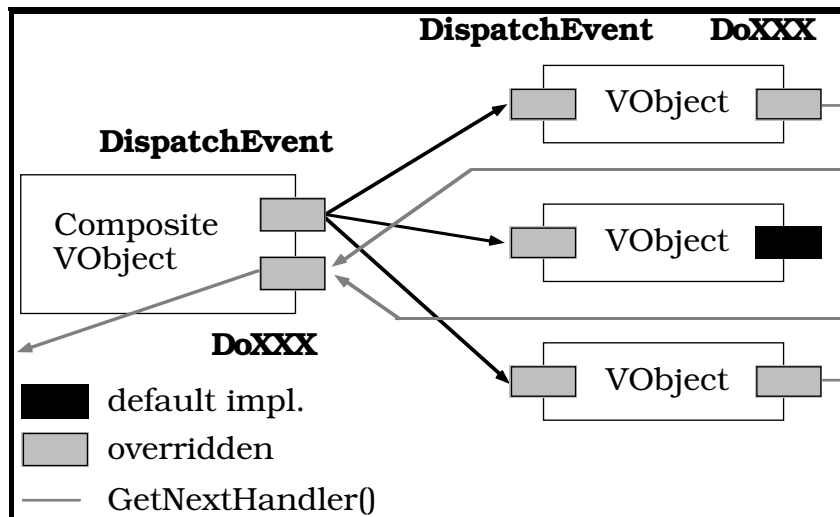
is the root class of all graphic classes and is responsible for

- layout management
- rendering on the screen
- event handling

A VObject object receives all events if the mouse is pressed in its bounding box.

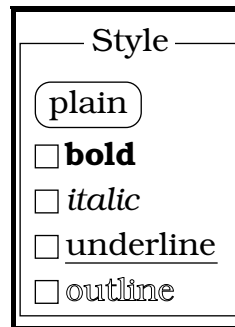
The method `DispatchEvents` then calls the appropriate dynamically bound method

(`DoLeftButtonDown`, `DoMiddleButtonDown`, ...)



GUI frameworks

Example:



```
new BorderItem("Style",
    new OneOfCluster(eVert, // OneOfCluster is a subclass
                        // of CompositeVObject
        new Button("plain", cldPlain),
        new ManyOfCluster(
            new Toggle("bold", cldBold),
            new Toggle("italic", cldItalic),
            new Toggle("underline", cldUnderline),
            new Toggle("outline", cldOutline),
            0
        ),
        0
    )
)
```

GUI frameworks

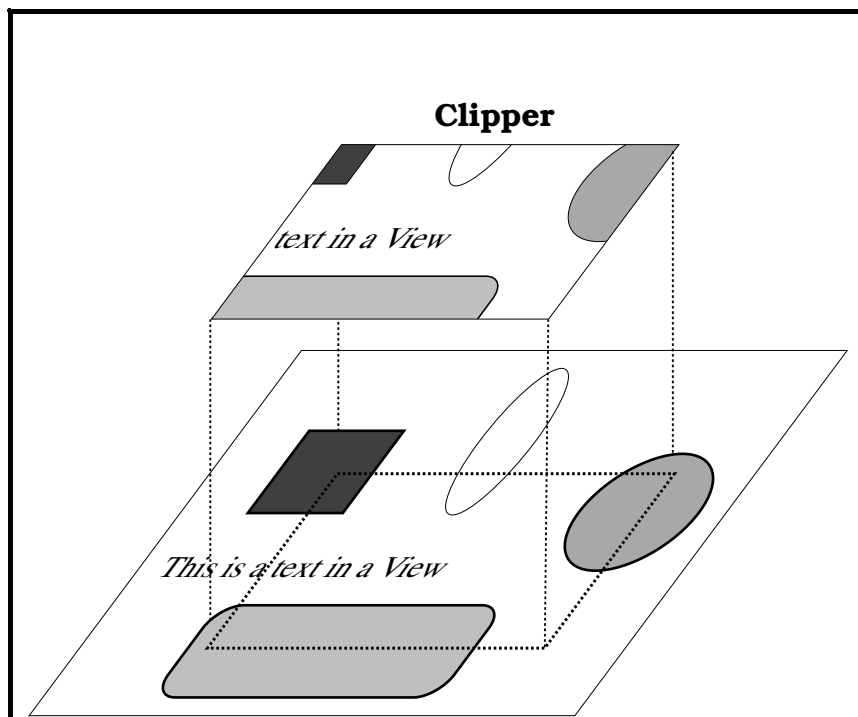
Graphic Building Blocks / Interaction Classes:

- class Clipper

A Clipper establishes a clipping boundary and an own coordinate system.

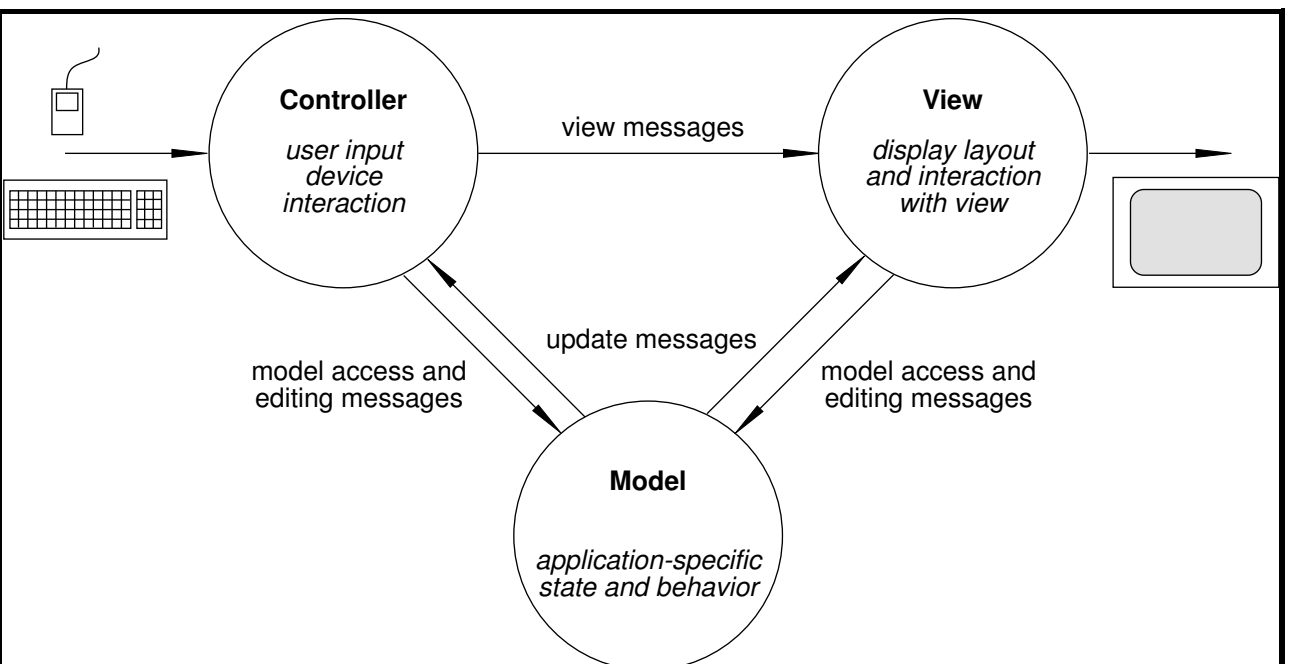
One built-in functionality of Clipper objects is the scrolling behavior (including autoscrolling).

A Clipper object clips any VObject object (abstract coupling!). As Clipper is itself a subclass of VObject, it is possible to nest Clipper objects to arbitrary depth.



GUI frameworks

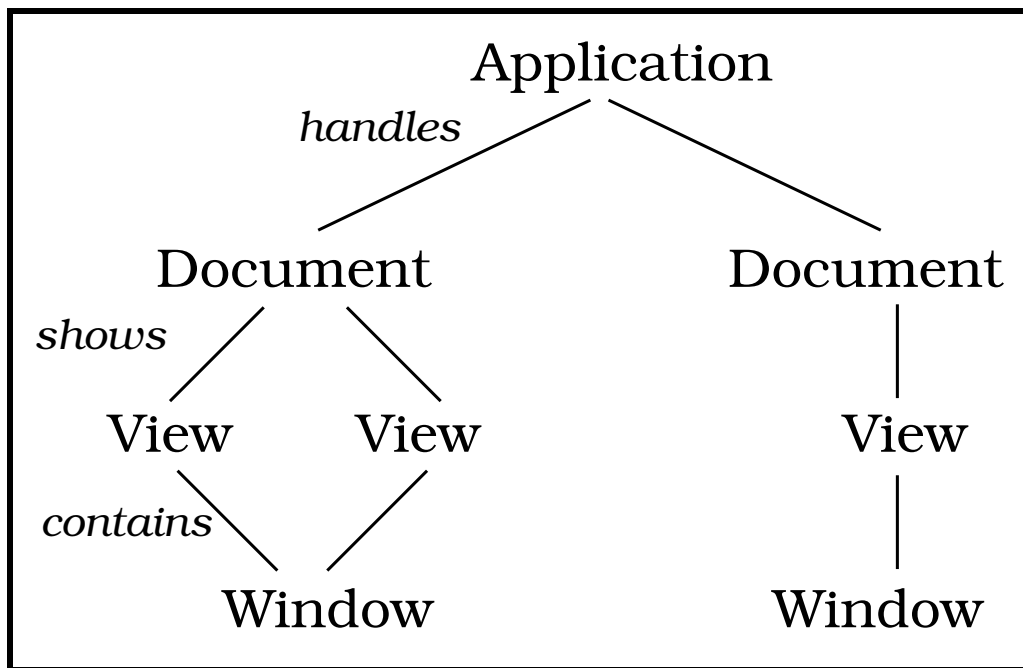
ET++ Application framework classes evolved from the original Smalltalk MVC-framework:



GUI frameworks

ET++ Application Framework Classes

Memory snapshot of an ET++ application at run-time:



MVC-framework versus state-of-the-art GUI frameworks:

Model + Controller → Document

View + Controller → View

GUI frameworks

- `class Application`

An instance of class `Application` controls the overall control flow and global states of the application.

`Application` handles its documents: an application can have several documents at any time.

Document objects can have different types—`Application` controls, whether it can load a specific type of document.

Important hook-method (= dynamically bound method that has to be overridden in a concrete subclass):

`DoMakeManager()`

This method has to be overridden to create application specific documents, i.e., objects of a subclass of the abstract class `Manager`.

GUI frameworks

- class Document

Document objects contain an application's data and a mechanism (i.e., dynamically bound hook methods) to store data on disk and to read them back into memory.

This mechanism is an example of how control flow can be factored out into an abstract class: **Document**, for example, implements the prompting of a proper dialog that asks the end user if he wants to close a document without saving it.

Another property of **Document** is that it cares for the display of its data by generating and managing the appropriate objects (**Window** and **View** objects).

GUI frameworks

- class `View`

is a subclass of `VObject` and factors out the control flow to display `VObject` instances and to print them. There exist some predefined subclasses of this abstract class:

CollectionView: Renders a collection of `VObjects` in a matrix -> useful for menus, menubars, scrollable lists.

TreeView: Renders a collection of collections of ... of `VObjects` as a tree.

GraphView: Renders a collection of collections of ... `VObjects` as a graph.

DialogView: Renders a tree of dialog items (i. e., a tree of `VObjects` and `CompositeVObjects`) as a dialog box.

TextView: Renders a `Text` data structure