

Framework design using inner classes - Can languages cope?

Kasper Østerbye and Thomas Quistgaard
IT University of Copenhagen
Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
kasper@itu.dk, tqvistgaard@itu.dk

ABSTRACT

Inner classes have been part of the Java specification since version 1.1, and are an integral part of the Beta language. In Java they have *primarily* been used in connection with event handling in the user interface libraries. This paper investigates inner classes as the cornerstone in the architecture for the layout part of a GUI framework, and how the two languages support this architecture. The difference in support in the two languages is shown to have clear impact on the usability of the framework for application programmers. While Java has come a long way, it turns out that three obstacles need to be removed to fully support the architecture. Of these, it should be straight forward to address two of them in Java. The third lies in the realm of aspect oriented programming. The proposed architecture is itself interesting, as it provides an insight into larger-scale use of inner classes, and provides a compiler supported idiom for the implementation of the composite design pattern.

1. BACKGROUND

In object oriented programming languages which support inner classes, e.g. Simula [Dahl *et al.*, 1968], Beta [Madsen *et al.*, 1993] and Java [Gosling *et al.*, 2005], we have come across an interesting implementation idiom which relates to the composite pattern [Gamma *et al.* 1995]. The design is that the lexical nesting mirrors the composition of the objects. A simple example in Java is:

```
class Menu{
    private String name;
    private List<Item> items = new ArrayList<Item>();
    public Menu(String name){
        this.name = name;
    }

    protected abstract class Item{
        private String name;
        public Item(String name){
            this.name = name;
            items.add(this);
        }
        abstract void action();
    }
}
```

The important part is that the class Menu has a list of Items, and each Item adds itself to the menu when created. The usage of generic collections does not play any role in the discussion. The class Item is declared protected, so it can only be used in subclasses of Menu.

The declaration of enables the following client code:

```
Menu editMenu = new Menu("Edit"){
    Item copy = new Item("Copy"){ void action(){...};
    Item cut = new Item("Cut") { void action(){...};
    Item paste = new Item("Paste") { void action(){...};
}
Menu fileMenu = new Menu("File"){
    Item quit = new Item("Quit") { void action(){...};
}
...
}
```

The implementation uses anonymous inner classes as a concise implementation of the singleton. In addition, lexical scoping avoids parsing a menu as parameter to items when they are created.

The idiom enables a somewhat declarative style, where the physical structure of the menu is mirrored in the program layout itself. Other examples of this structure are the relationship between rule-set and individual rules, where the rules can be defined inside their rule-set; or the hierarchical structuring of a GUI, to which we will return. In Beta, we have also applied the idiom in the area of process composition [Østerbye & Kreutzer, 1999].

In general, there are a number of qualities which one would like a framework to have:

- It should be simple to use for the application programmer
- Misuse of its constructs should be captured at compile time
- Its application should be concise
- The framework should be extensible
- The underlying implementation should perform adequately – that is, should not constitute a bottleneck in the application

Also, it is important to realize that a framework is a generalization over a *set* of applications. There are therefore (interesting) applications that are covered and other that are not covered by the framework.

Compared to the above qualities, the simple menu illustrates some important points:

- Simplicity. The application programmer need not have an explicit set of statements which associates items to menus. The menu structure is manifest in the program structure.
- Compile-time checks. Attempting to use Items outside the scope of a Menu will not work, class Item has been declared protected, and can therefore only be seen in subclasses of Menu. The compiler checks this (but will give un-informing error-messages in case of violations).

- Conciseness. There is not much extra information except the definition of the hierarchical structure between Menu and Item. There is some redundancy (Item and item name repeated twice), which we will return to.
- Lack of flexibility. These qualities have been obtained at the cost of not being able to dynamically change which menu a given item belongs to.

In the remainder of this paper this idiom will be further elaborated. The next section introduces the framework we have developed to investigate the idea. The description highlights the inner class idiom, and the problems we have encountered in implementing the idiom in Java. Then we contrast some of the problems encountered in the Java solutions with similar (but less problematic) solutions in Beta. We end with a summary of our findings.

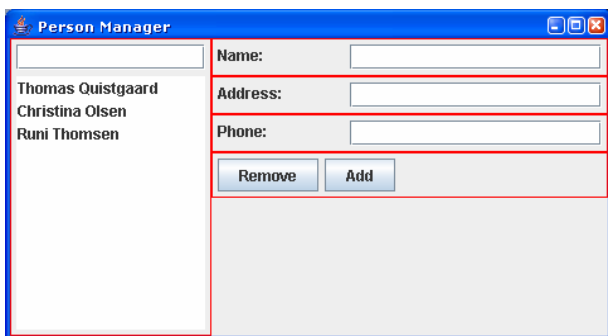
2. HIERARCHICAL GUI LIBRARY (HGL)

Before examining the differences between the languages, a slightly more complex example is needed. As part of his master's thesis, Thomas Quistgaard [Quistgaard, 2005] designed and implemented a hierarchical user interface framework. The goal was to apply the above inner class idiom for a full scale framework to achieve a simpler to use GUI framework than say Swing, which is notorious for its complexity.

The design is based on a few overall guiding principles:

- The program structure should mirror the hierarchical structure of the GUI.
- The components are added to their enclosing container in the order they are declared (Using the same idiom as with the menu).
- The physical layout is declared using annotation types, to provide a clear separation between hierarchical structure and physical layout, and to provide a path for later tool manipulation of physical layout.

To explain the design, a simple example will be used.



The above Frame (top level window) has to its left a text field in which one can enter a search string. All persons that contain the string in their name are shown in the list below. Selecting a person brings up the underlying data in the right part for examination or modification. The above GUI is defined in HGL as shown below.

This example illustrates the three design principles. Every graphical object which appears inside the frame is declared as member fields of the anonymous personManager Frame. Anonymous inner classes give a syntactic structure which enables the hierarchical

structure to follow the program structure. Inside the frame, a panel is declared, inside which a text field and a list are declared.

The components are added in the order they are declared. Inside list-panel, a TextField, and then a List are declared. These are added to the panel in the order of declaration.

```
Frame personManager = new Frame("Person Manager") {

    @Vertical
    Panel listpanel = new Panel() {
        @Width(150)
        TextField searchtextfield = new TextField();
        @Width(150) @Height(300)
        List list = new List(); // GUI list, not a collection library List
    };

    @Vertical @Padding(0)
    Panel infopanel = new Panel() {

        @Horizontal
        Panel namepanel = new Panel() {
            @Width(100)
            Label namelabel = new Label("Name:");
            @Width(200)
            TextField nametextfield = new TextField();
        };

        @Horizontal
        Panel addresspanel = new Panel() {
            @Width(100)
            Label addresslabel = new Label("Address:");
            @Width(200)
            TextField addresstextfield = new TextField();
        };

        @Horizontal @Hlock(false)
        Panel phonepanel = new Panel() {
            @Width(100)
            Label phonelabel = new Label("Phone:");
            @Width(200)
            TextField phonetextfield = new TextField();
        };

        @Horizontal @Hlock(false)
        Panel addpanel = new Panel() {
            Button removebutton = new Button("Remove");
            Button addbutton = new Button("Add");
        };
    }; // end infoPanel
}; // end personManager
```

Frames and Panels are containers that contain other components, including other Panels. Layout is defined using annotations. Annotations are user defined metadata. Syntactically, annotations are located as modifiers, in front of the element they annotate. Annotations are accessed programmatically through reflection. @Horizontal is a user defined annotation, which is used to specify that the layout in the panel should be horizontal instead of the default vertical. Annotations can include simple values as parameter. @Padding indicates the space between a component and the previous component in the same container (or the border if it is the first). Hlock and Vlock indicate resizing behaviour.

2.1. Addressing components

The above code does not specify behaviour, only layout. There are two kinds of behaviour which is interesting in connection with GUI frameworks: tying the GUI to the application data and business logic, and

(our focus) ensuring graphical consistency. If we select “Christina Olsen” and modify her name, that name change ought to be reflected not only in the application data, but also in the list. Standard Swing list listeners raise an event if an element is being added or deleted from a list, but not if an element is changed. A direct approach would be to let the `textChanged` event from the `nametextfield` directly change the list:

```
TextField nametextfield = new TextField(){
    void onChange(TextChangeEvent e){
        listpanel.list.changeSelected(this.getText());
    }
}
```

But this does not work because `listpanel` is of type `Panel`, and `Panel` does not have a field named `list`, though the concrete object `listpanel` refers to does indeed have this field. To get around this, we implemented a method `get` (using reflection), which allow us to write the above code as:

```
TextField nametextfield = new TextField(){
    void onChange(TextChangeEvent e){
        ((List)get("listpanel.list")).changeSelected(this.getText());
    }
}
```

Unfortunately, we can no longer check at compile-time that the path exist and is spelled correctly.

2.2. Compile-time checking

The hierarchical definition of the components plays an important role in making certain that the compiler can catch as many mistakes as possible.

At the outset, the design looks like a composite pattern, with the components as leafs, and `panel` and `frame` as composites. In Swing, a `Frame` is a top-level window, and as such:

- No component exists outside a frame
- No frame can be put inside a frame (a frame is not a component)

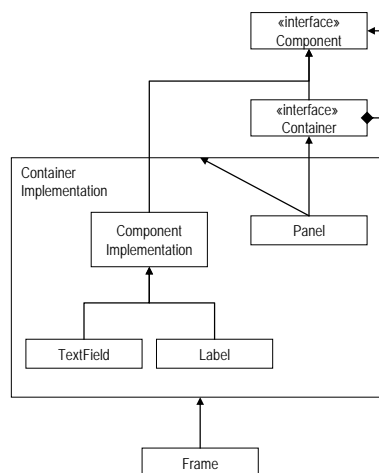
The standard composite pattern does not treat the issue of a dedicated root composite. In particular, no compile time checks are carried out.

To provide a compile time checkable version of the rooted composite pattern, we can again use inner classes as a key:

```
interface Component{...}
interface Container extends Component{
    List<Component> getComponents();
    void addComponent(Component c);
}
class ContainerImplementation {
    private List<Component> components;
    public Component getComponents(){ return components; }
    public void addComponent(Component c){ components.add(c); }
    protected class TextField implements Component{...}
    protected class Label implements Component{...}
    protected class Panel extends ContainerImplementation
        implements Container{...}
}
public class Frame extends ContainerImplementation{
    ...
}
```

The interface for components has a specialized interface which represents containers. The container interface specifies that it

consist of components, whereby we obtain the usual recursive composite pattern.



The class `ContainerImplementation` contains the necessary infrastructure to implement the `Container` interface, but does not declare that it does so (no `implements` clause). `ContainerImplementation` has two subclasses – `Panel` and `Frame`. `Frame` is a public class, and can be used as expected. However, it does not implement the `Container` interface; hence `Frames` are not components and cannot be contained in a container. `Panel` on the other hand declares that it implement the `Container` interface. The methods to do so are inherited from `ContainerImplementation`.

Leaves (e.g. `TextField` and `Label`) and `Panel` are protected inner classes of `ContainerImplementation`, and can therefore only be used in subclasses of `ContainerImplementation`, whose only public subclass is `Frame`. Within a concrete `Frame`, e.g. the anonymous class assigned to `personManager`, one has access to the protected inner classes `TextField`, `Label` and `Panel`.

The design does not change the way how the application programmer uses the framework. And it achieves the two compile-time checks we wanted:

- It is not possible to add a frame inside a frame, as a `Frame` is not a component.
- It is not possible to use any components outside a frame, since they are protected inner classes of `ContainerImplementation` (allowing components to be used both within `Frame` and `Panel`).

We find this idiom for implementation of the composite design pattern a contribution in its own right. It gives a solution to the notion of a special root composite, and it can enforce this design at compile-time.

3. IS BETA BETTER?

The original idea for the HGL framework originates in Beta [Lidskjäl, 2002]. In [Quistgaard, 2005] the design is expanded, in particular by adding declarative layout, and accommodating the design to fit Java. In this section we will examine a few issues where Beta and Java differ and how this impact the details of the library.

3.1. Variable declaration syntax

A mundane difference between Java and Beta relates to how variables are defined. Java declares variables as “Type varName”, whereas Beta does “varName: Type”.

In Beta, the declaration of `editMenu` would look like (Though we use `{...}` instead of Beta’s `{#...#}`):

```

editMenu:@Menu{
  copy:@Item{ action{...} };
  cut:@Item{ action{...} };
  paste:@Item{ action{...} };
}

```

The Beta syntax is more concise, as we avoid stating `Item` twice, both as type name and after new. Furthermore, we do not need to pass the name of the menu item as parameter, as we are able to pick out the name through reflection.

A more important consequence of the different way of declaring the variables surfaces in connection with the addressing of components. In Java, it was necessary to use reflection in addressing components, although we were able to hide this in the `get` method. In Beta, the type of the corresponding `listPanel` variable is the actual type which does have a `list` field. Hence, in Beta we are able to avoid reflection in connection with field addressing.

3.2. Reflection

Reflection is typically avoided because of bad performance and because it postpones checks to runtime. In the concrete design of HGL, event handling was not done as described earlier. Instead a mapping between events and handler methods were established programmatically as:

```
((List)get("listPanel.list")).onSelect("listElementSelected");
```

A series of such statements are executed at initialization. The `get` method returns a list, and this list is told what method to execute when an element is selected.

Because of reflection, the spelling error "`listPanel.list`" (should be "`listpanel.list`") is first caught at runtime. However, the mistake is caught at program initialization. Thus, running the program just once will reveal the error. Efficient run-time structures are constructed during initialization, so no execution time is lost in practical usage.

3.3. Annotation checks at compile time

It would have been compelling to use the new annotation processing tool [APT 5.0] to check that annotations were used correctly, for instance that the `Vertical` annotation is only associated with `Panels`. However, neither Java reflection nor APT allows us to access inner classes; hence we cannot traverse structures of anonymous inner classes. So while we basically have all the machinery in place, a design choice in Java and APT prevents us from doing compile-time checks in connection with applications of our framework.

3.4. Object initialization

Java has two ways to provide information when an object is created. One can pass parameters to its constructor, and sometimes one can attach annotations to its declaration. In Beta one cannot do either one.

So, the best approximation one can do for the layout information in Beta is something like the following

```
listPanel:@Panel{ Layout::Horizontal; Padding::(do 0->padding);
...
}
```

This syntax specifies that `listPanel` is a constant which refers to an object which is a subtype of `Panel`, where the virtual type `Location` is bound to `Horizontal`. This corresponds roughly to giving

`Horizontal` as a type parameter. The method `Padding` is specialized to return the value `0`.

There are several drawbacks compared to the annotation approach

- Annotations are well suited for tool manipulation.
- The specification of concrete values, like `0` padding becomes quite clumsy.

But there are a number of drawbacks associated with the annotation approach we have used as well.

- 1) One cannot associate annotations with anonymous inner classes. Hence we have been forced to annotate the fields instead.
- 2) Annotations need to be manipulated through reflection, which implies poor performance. In our case, however, it is only done when the `Frame` is initialized, not when the `GUI` is used.

3.5. Framework extension

The complex design makes it hard to add new component types to the library, as we effectively need to add new definitions inside a package protected class. The Beta compiler supports a `Fragment` system. The `fragment` system is a way to declare insertion points in classes, and enables libraries of code which, at compile-time, is weaved into these insertion points. A problem in HGL is that one cannot add new protected component types to the `ContainerImplementation` class. A Java version of Beta's `fragment` system would allow us to write the `ContainerImplementation` class as:

```

class ContainerImplementation {
  private List<Component> components;
  ...
  protected class Panel extends ContainerImplementation implements Container{
    ...
  }
  «SLOT ExtraComponents: Declarations»
}

```

A component, e.g. `GanttChart`, can be written, specifying that is intended to be inserted at the `ExtraComponents` slot. `GanttChart` is compiled as if it were lexically located at that slot, with access to all the same lexical information as the standard components.

To use `GanttChart`, in your application, you declare it in an insert clause. Rather than making `GanttChart` available in the global name space, `insert` makes `GanttChart` available as if it were inserted into the slot. Hence, `GanttChart` cannot be used outside of `Frames`. On the other hand, it is readily available to be used as any other components.

It is highly unlikely that such a mechanism should be included in Java. A similar effect can be obtained using aspect oriented programming. The idea is to use insertion to place the `GanttChart` into the `ContainerImplementation` as:

```

aspect MyComponentLibrary {
  protected class ContainerImplementation.GanttChart {...}
}

```

At present, however, the most widely used aspect compiler for Java, `AspectJ` [AspectJ, 5.0] does not support insertion of inner classes, and aspect oriented programming tends to focus on other issues than insertion. The difference between the slot approach and aspects is discussed in [Ernst, 2000]. For our needs there is no fundamental difference.

While `C#` does not support inner classes, its notion of partial classes is also a solution. If `ContainerImplementation` were partial, it could be extended with new components. Partial inner classes have to be worked out in practice. The slots in Beta can only be used for adding new classes and

methods, not new fields, as that would change the size of objects, which would prevent separate compilation.

4. SUMMARY

With some tradeoffs, we have been able to implement HGL in Java. Its design, however, is cleaner in Beta. In particular we have encountered three major problems in Java:

First, the problem with the type of variables and anonymous inner classes in Java is a hindrance for our design of HGL. One solution is to adopt the `val` type from ML, to state that the type of a variable should be deduced by type inference. Hence, our `listpanel` should be defined as:

```
final val listpanel = new Panel(){
    ...
    List list = new List();
}
```

This way the type of `listpanel` could be the anonymous subclass of `Panel` which has the field `list`, so it can be compile-time checked that `listpanel.list` is indeed a legal object path.

While it is unknown if such a `val` construct will make it to Java, a variation which can solve the problem will be available in next version of Visual Basic.

Second, the standard java annotation processing tool allows us to write our own compile-time modules. This facility is intended for writing code-generators in connection with J2EE. However, it is tempting to view it as general compiler extension mechanisms, which allow us to write custom compile-time checks for the usage of libraries and frameworks. In its present state, however, we cannot use it for HGL. Nevertheless, a possible example might be the unit testing framework JUnit [JUnit]. JUnit assumes certain naming conventions, which are checkable using reflection, and can also be checked at compile time. We have not investigated this further. But in our case, neither reflection nor APT allows us to examine the whole program; in particular anonymous inner classes can not be traversed.

Thirdly, to make the inner class approach presented here feasible, it is necessary to solve the problem of adding inner classes to an existing class. Java needs to be extended with something similar to partial classes, or AspectJ needs to be able to handle introductions of inner classes. The notion of MixIn Layers [Smaragdakis & Batory, 1998] provides another view on how the existing framework can be refined into a new framework with additional components. Their solution provides the necessary infrastructure we ask for, but from our experience with HGL we do not necessarily need all the capabilities of MixIn Layers.

Scala [Odersky *et al.*, 2005] provides the key mechanisms needed to implement the inner class idiom as well, in particular object definitions and anonymous inner classes. However, it seems that Scala has the same problem as Java when it comes to extending the framework, and it is not clear what mechanisms can be used to separate logical and physical layout.

Compared to [Hedin & Knudsen, 1999] we are applying some of the mechanisms from Beta that they describe as providing benefit for framework design. In relation to their work, the contribution in this paper has been to apply those guidelines in the context of a Java based framework, and to report where Java fails in achieving the goals. However, an important issue for framework design not mentioned in [Hedin & Knudsen, 1999] is object initialization.

Here Java is superior to Beta, providing both field initializers and annotations.

Of the major object oriented languages, it is only Java that supports inner classes. C# and C++ share a design, in which a class can be defined inside an other class, but the inner class will not have instances of the outer class as lexical scope for its objects, hence not even the simple Menu-Item example will work. Eiffel, Smalltalk and many other languages do not even allow the simple nesting of C++ and C#.

5. REFERENCES

- [APT 5.0] Annotation Processing Tool (apt), part of Java 2 Standard Edition. <http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html>
- [AspectJ, 5.0] AspectJ Project. <http://eclipse.org/aspectj/>. Accessed October 3rd, 2005.
- [Dahl *et al.*, 1968] O.J. Dahl, B. Myrhaug, K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center, Oslo, 1968.
- [Ernst, 2000] Erik Ernst Syntax based modularization: invasive or not? in Tarr, P., Bergmans, L., Griss, M. and Ossher, H. (eds.), *Workshop on Advanced Separation of Concerns (OOPSLA'00)*. Department of Computer Science, University of Twente, The Netherlands.
- [Gamma *et al.* 1995] Eirich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Gosling *et al.*, 2005] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification (Third Edition)*. Addison Wesley 2005.
- [Hedin & Knudsen, 1999] Görel Hedin and Jørgen Lindskov Knudsen. Language Support for Application Framework Design. In *Implementing Application Frameworks: Object Oriented Frameworks at Work*. Ed. M.E. Fayad, D.C. Schmith, R.E. Johnson. Wiley 1999.
- [JUnit] Unit testing framework for Java. <http://www.junit.org/index.htm>
- [Lidskjäl, 2002] *Lidskjäl: User Interface Framework – Tutorial*. Mjølner Informatics Report, MIA 95-30, February 2002. http://www.daimi.au.dk/~beta/mjolner_system/lidskjalv.html
- [Madsen *et al.*, 1993] Ole Lehrman Madsen, Birger Møller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley 1993.
- [Odersky *et al.*, 2005] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification Version 1.0*. Accessed from <http://scala.epfl.ch/docu/index.html> October 3, 2005.
- [Quistgaard, 2005] Thomas Quistgaard. *The Hierarchical Graphical Library*. Masters Thesis, IT University of Copenhagen, 2005. <http://hgl.sourceforge.net>. (in Danish)
- [Smaragdakis & Batory, 1998] Yannis Smaragdakis and Don Batory. *Implementing Layered Designs with Mixin Layers*. Proceedings of ECOOP'98, Brussels, Belgium, July 1998. Lecture Notes in Computer Science 1445, Springer-Verlag.
- [Østerbye & Kreutzer, 1999] Kasper Østerbye and Wolfgang Kreutzer. *Synchronization abstraction in the BETA programming language*. Computer Languages 25 (1999) 165-187.