

The Framework Use Problem: A Preliminary Study With GUI Frameworks

Daqing Hou, H. James Hoover, and Changyu Yin
University of Alberta
Department of Computing Science
Edmonton, Alberta Canada T6G 2E8
{daqing, hoover, eunice}@cs.ualberta.ca

Abstract

This paper reports our study of the problems associated with the use of two GUI frameworks, MFC and Swing. We argue that learning a framework is a design recovery process; the process is further characterized into several aspects. We also discuss a few techniques that can potentially ease the learning curve. The effectiveness of these techniques is explained intuitively using the '7+/-2' principle. Examples are drawn from the Swing JTree component. We expect our paper to serve both as a data point for illustrating the framework use problem and as the basis for further study in this area.

1. Introduction

Object-oriented frameworks are a moderately successful approach for leveraging proven software designs and implementations to reduce the cost and improve the quality of software. In contrast to earlier reuse techniques based on function and class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). When combined with components, frameworks provide the most practical large scale reuse [5].

It has been known, however, that frameworks are often hard to learn and use. Although this may sound unusual at first, one should not take it as a surprise, if we realize how much information that any real frameworks can convey. Actually, we conjecture that for any framework, there always exists a lower bound on the amount of effort that is required to understand it; that consists of its essential complexity. We do not know much about the nature of the lower bound yet, but we observe that in reality, average programmers are spending lots of effort in order to use a framework. This makes the research of the framework use problem both a practical and important topic.

Much research effort has been dedicated to mitigating the problem, for instance, by refactoring frameworks to 'blackbox' ones [12], by proposing new programming constructs such as the mixin layers [2] and compositional contracts [6], and by improving documentation such as the hooks method [4]. But all above but hooks attack the problem by inventing new building techniques. The claimed benefits have not been validated in practice yet, and these techniques have not become mainstream practice. Moreover, it seems that there are not much detailed studies in the literature that illustrate the problem with concrete examples.

This paper reports some lessons learned from two recent projects. The first two authors have spent some time in studying the design of the MFC framework; the primary goal of that effort is to understand both how a large OO framework is designed and the nature of the problems faced by the programmers who use it. Following that effort, we conducted a four month project to collect and analyze the questions that programmers posted to the Java Swing framework, specifically, those related to the JTree component.

The following summarizes the main points of the paper. To learn a framework is to learn its design. The design recovery process can be characterized as follows:

- The amount of information can be overwhelming.
- Understanding collaboration is the most difficult task.
- The design method of decomposing large classes into smaller ones makes customization easier but learning more difficult.
- Optional features further complicate the issue.

We also find that for complex hot spots, framework designers can help the future users by *prioritizing features and providing appropriate default implementations*.

The framework use problem is a multi-faceted one, where many factors can come into play. For instance, design quality is of course the most important factor; without a good design, it is unimaginable how one can use it. Even if a framework possesses a good design, there still remains the challenge of teaching programmers how to use it; this is essentially an education problem. And last but definitely not the least, the knowledge levels of the programmers are also relevant. Clearly, our result here can only expose a small corner of the iceberg. Particularly, this paper does not provide any solutions, although it does discuss some possible routes that may lead to them. Instead of as a solution paper, it'd better be viewed as a data point for further study.

The rest of the paper is organized as follows: first the next subsection reviews the '7+/-2' principle. Section 2 provides some characteristics of the process of learning frameworks. Section 3 discusses some directions that could improve the current state of practice. Section 4 concludes the paper and discusses future work.

1.1. The '7+/-2' Principle

To lay a solid foundation for our further discussion, we first review the '7+/-2' principle [11], a result from Psychology. Briefly, the principle reveals that there is a limitation on the number of distinct tokens that human beings are able to receive, process, and remember at a time, which is usually at the neighbourhood of 7. To break (or at least stretch) this informational bottleneck, we can either organize our subjects into dimensions or utilize abstractions to recode successive subjects into a sequence of chunks.

2. Some Characteristics of Learning OO Frameworks

The goal of learning is to understand the design of the frameworks. Ideally, we would expect that the frameworks hide from the programmer as much detail as possible so that he would only spend time on the necessary artifacts, no more and no less. Unfortunately, in practice, we do not

yet have a well-defined method to achieve this. Sometimes a programmer even has to look at the source code of the framework to solve a particular problem. Like the way that searching algorithms work, program understanding involves exploring unknown branches and backtracking, too often some of which turn out to be irrelevant to the current problem.

2.1. Learning a Framework Is a Design Recovery Process

The following shows two examples on how the design is rationalized.

Looking more closely at the difference between the `TreeModel` interface and the `DefaultTreeModel` class, we find that the class defines a group of ‘change’ methods and ‘fire’ methods. A natural question to ask then is why these methods are not included in the `TreeModel` interface. The answer lies in the fact that if they were included, any class that implements the `TreeModel` interface would also have to implement them. For ‘read-only’ trees, however, there is no need of these methods, thus it would be an extra burden for the programmer of a ‘read-only’ tree to implement these methods. We suspect that this is the major reason why the `JTree` designers have left these methods out of the interface.

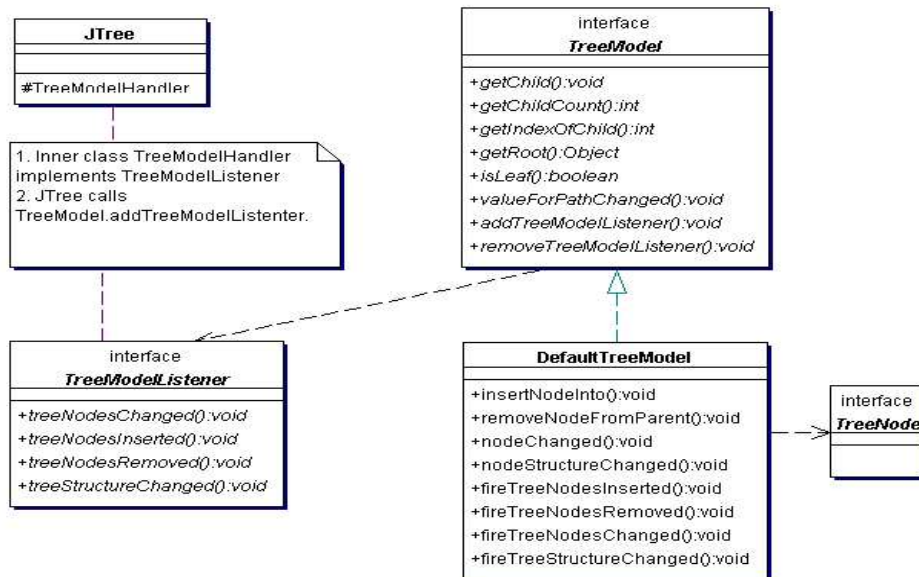


Figure 1. The class diagram for the `JTree` component

Since there are no changes to a read-only tree, we start to wonder why `addTreeModelListener` and `removeTreeModelListener` appear in the `TreeModel` interface. We find that this is actually a technical compromise. As an implementation of the Observer pattern, in theory, `TreeModel` can have multiple observers registered to it. Each of these observers is required to implement the `TreeModelListener` interface. The `JTree` class is just one of these observers; it turns out that `JTree` has a protected inner class `TreeModelHandler` that implements the `TreeModelListener` interface. When a `JTree` object is created, it will automatically create a `TreeModelHandler` object and register the object to its `TreeModel`. Whenever the listener is notified that the tree model has been changed, it will update the tree picture associated with its `JTree` object accordingly. Apparently, the appearance of `addTreeModelListener` and `removeTreeModelListener` in the `TreeModel` interface is just to

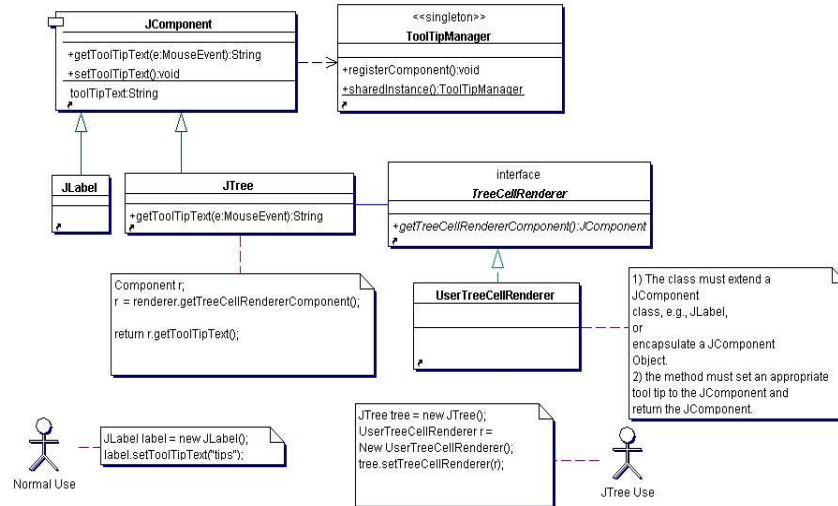


Figure 2. The class diagram for the tool tip feature

make the implementation of the JTree class possible.

2.2. The Sheer Size of Information

The size of information is manifested on several aspects:

1. Many classes. For instance, there are 800 or so classes in the Java Swing.
2. Many methods and properties. For examples, the JTree class has about 400 methods, 120 of them are defined by JTree class itself, and 280 of them are inherited from three of its super classes, that is, javax.swing.JComponent, java.awt.Container, and java.awt.Component.
3. Methods with similar names or meaning are difficult to differentiate. For instance, the ‘fire’ and ‘change/insert’ methods in the JTree example.
4. Redundancy in the design, such as more than one way to do a thing, requires more effort for the programmers to learn.

This list is by no means complete, and much more can be added. But the point is that any non-trivial framework will necessarily generate large amount of information for its users.

2.3. The Difficulty of Tracing and Chunking

When reading the source of an OO framework, programmers do tracing and chunking. The goal is to find all the relevant classes and interfaces, methods, and attributes to form a coherent mental model on how the artifacts work together to achieve a certain functionality.

Due to the way that OO software is structured, however, tracing and chunking can be a hard work. Often a feature of the framework is implemented by several methods that distribute over multiple classes, some of which are implemented on different layers of the inheritance hierarchy while others by the associated classes. The following quote from an MFC news group can illustrate the problem better:

“This (when or if you should call the base class version (of functions) from within your overridden version? ... And if you do call the base class version, should you call the base class version before the code in your version or after) is (what) I think one of the most confusing things about MFC... I really think the documentation for all these `CWnd` virtual functions and message handlers should spell this out clearly, case by case. As it is, the only way is to look at the source code for the base class, but since these functions may or may not be implemented at each level of the class hierarchy it is not that easy.”

This difficulty can be explained by the ‘7+/-2’ principle. When a programmer is understanding a framework, the number of distinct program elements that he/she faces can grow very fast. The programmer has to not only find the relevant elements and rationalize them, but also exclude the irrelevant ones. It is easy to get overwhelmed by doing this. One way to appreciate the complexity is to simply count the number of program elements in Figure 2: there are 5 classes and interfaces and 6 methods, not to mention their relationship and the other methods within the classes and interfaces.

2.3.1 An Example: Set Tool Tips for Tree Nodes

In Swing, the tool tip functionality is implemented as an Observer pattern. The singleton class `ToolTipManager` observes the mouse actions of any GUI component that needs a tool tip. In order to show a tool tip, a component has to register itself to the `ToolTipManager`, which in turn registers itself as an observer of the mouse actions of the component. Once a component is registered, whenever the mouse enters its display area, the `ToolTipManager` will start up a timer for it. If the mouse leaves the component before the timer times out, then nothing happens; otherwise, `ToolTipManager` will send a `getToolTipText` message to the component to ask for a tool tip and then display the text over the component.

The `JComponent` class has two methods related to tool tips. The first void `setToolTipText` (string text) sets a tool tip to the component and the component will store it as a property; this method also calls `ToolTipManager.sharedInstance().registerComponent(this)` to register the component to the singleton `ToolTipManager`. However, if text is null, then this method will unregister the component from `ToolTipManager`. The second String `getToolTipText()` returns the tool tip.

Ordinary GUI components like `JLabel`, `JButton`, and `JPanel` etc can be easily set up a tool tip using the two methods of `JComponent`. However, in order to display a tool tip for a complex component like `JTree`, a programmer has to know more. A tree can consist of multiple tree nodes. `JTree` uses a `TreeCellRenderer` object to render tree nodes. More specifically, `TreeCellRenderer` has a method `getTreeCellRendererComponent` which returns an object that can be used to draw a tree node. By default, a `TreeCellRenderer` object is derived from `JLabel` and its `getTreeCellRendererComponent` method returns *this*. Thus, the returned drawing object actually inherits `JLabel`'s drawing methods to draw a tree node on the screen. However, the tree nodes that we are seeing on the screen are not really GUI components since the drawing object is not added to the GUI containment hierarchy and it cannot receive any event. This also implies that it is impossible to register the drawing object to the `ToolTipManager` and set up tool tips in the normal way.

`JTree` implements tool tips by overriding `getToolTipText(MouseEvent event)`: Based on the information carried with the parameter event, the method first gets the tree node that needs a tool tip;

No. of classes involved in solutions	No. of problems (out of 180)	Percentage
1	22	12.22
2	70	38.89
3	45	25.00
4	31	17.22
5	11	6.11
6	0	0.00
7	1	0.56

Table 1. The numbers and percentage of problems involving different number of classes

it then sends a message `getTreeCellRendererComponent` to the `TreeCellRenderer` object with the node as an argument; as a result, the message returns a drawing object. After getting the drawing object, the method sends the `getToolTipText` message to it and returns the obtained text.

With the above design information in mind, a programmer can have two options to implement tool tips for `JTree`:

1. If a uniform tool tip text is applicable to all tree nodes, he must (1) register the `JTree` to the tool tip manager by calling `ToolTipManager.sharedInstance().registerComponent(theJTreeObject)`; (2) override `JTree`'s `getToolTipText` simply returning the uniform text.
2. It is highly possible that most applications would need node specific tool tips. In this case, the programmer must (1) do the same as item (1) above; (2) override `TreeCellRenderer`'s `getTreeCellRendererComponent`. Inside the override, first determine a tool tip text for the node (which is one of the parameters of the method) and then set the text to the component by sending it a `setToolTipText` message. Do anything else in between and at last, return the component.

2.4. Understanding Collaborations Dominates the Task

“[...] no object is an island. All objects stand in relationship to others, on whom they rely for services and control.” [3] The implication of this quote is that typically, elaborating a hot spot needs to know multiple “slots” of the framework. According to the ‘7+/-2’ principle, the performance of understanding can become worse and worse when the number of distinct information that a programmer faces exceed a certain limit. Thus it should not come as a surprise that understanding collaborations consumes the largest percentage of the effort.

Our data support this view. We have collected 180 or so problems about the Swing `JTree` component and done some analysis with them. First, for each of the collected problems, we come up a correct solution. As a result, we get a table which shows the classes that each solution involves. Then we count the number of classes involved in each solution. At last, we count the number of problems which involve from 1 class to 7 classes. The result is shown in Table 2.4. From the table we can see that more than 85 percent of the problems involving two or more classes.

2.5. Decomposing Large Classes into Smaller Ones

OO design [8] encourages designers to decompose a large class into smaller ones. The main advantage is that, as a result, it becomes more flexible to adapt the behavior of the design. These

smaller classes can be customized independently and then composed with the other classes to form a collaboration. The `TreeCellRenderer` is such an example. The action listeners of the Swing framework are another example.

This design method, however, seemingly has some negative effect on the goal of ‘ease of learn and use.’ Now the programmer has to deal with more classes, knowing their existence, their locations, their role in the collaboration, and their relations. Because the information is distributed, it becomes harder to locate the pieces to form a complete big picture about the design. Even worse, many times it may not be obvious for a programmer to know the relevance of a class in a collaboration.

We are not against the method but simply point out its negative effect on understanding. As a rule of thumb, when trying to figure out the role of a class, the programmer should always look for the appropriate collaboration context, because such a class often participates in a collaboration, and its meaning can only be understood by knowing its context.

2.6. Optional Features

Informally, a feature represents a logical aspect of a framework that is meaningful to its users. It can be as simple as just one method or as complex as a number of classes and methods. In general, it is subjective to decide of what a feature is consisted.

Features of frameworks can be optional. For example, in Swing, setting tool tips can be considered as a feature of the GUI components. Many applications do not use tool tips at all. It is less often used than other features such as layout management and event handling. Moreover, the tool tip feature has another feature that allows one to change the elapse interval of the component (an elapse interval is a period of time. If the mouse remains in the component beyond that period of time, a tool tip will display). The latter feature is used much less often.

An optional feature add feature specific program elements into the source of a class. As a result, source code is turned into a mixing of different concerns. Apparently, it is very difficult for a programmer to recognize and distinguish between them by merely reading the source code.

3. Directions

This section discusses the techniques that can be useful to ease the learning task. They are categorized into three aspects of documentation, design, and experience, all of which can be explained in terms of the ‘7+/-2’ principle.

3.1. Improving Documentation

Object–Oriented frameworks pose unique challenges to documentation, which have been documented by several authors [9, 7, 4]. In the following, we emphasize the importance of having an architectural level understanding. We also suggest two specific techniques that are inspired by our experience.

Architectural level documentation is crucial. A good architectural description of a framework can greatly help to understand it. Without such a description or without a firm understanding of it, application programmers would be mired in details and confusion.

The role of architectures can be explained by the ‘7+/-2’ principle. An architectural level description is actually an abstraction of the underlying system. There can be two ways of doing

abstraction: one is to abstract out a common aspect of a group of things, the other is to highlight an important part of a thing. Either way, abstraction can help to decrease the number of information that a subject has to understand, by “encoding” them with a name. Once understood, the information can be remembered by the name rather than by the low level, specific details.

Software architecture is important because it can help us make easier perception of a software system. Software architecture is essentially abstraction and abstraction does not imply ambiguity or imprecision. A good abstraction conveys the essential elements and the key relations between these elements so that once learning it, a programmer can repeatedly apply them to a larger range of software artifacts. For example, given a description of a certain feature, a software architecture can guide us track down the modules that implement that feature.

Grouping and ordering documentation are also important. By grouping relevant things together and presenting them as a whole, we can save much for programmers. For example, instead of showing the method-by-method style of documentation, we can partition the 120 or so methods of the JTree class into groups like painting, tree expansion, tree collapse, tree model, and visual properties etc. According to the ‘7+/-2’ principle, this will be much easier for programmers to understand because they do not have to do the grouping by themselves. Similarly, documentation that tell a complete story is much more preferred than the JavaDoc-generated class by class, method by method documentation. It seems to be too expensive to recover a complete mental picture about the framework from the latter kind of documentation. In fact, each non-trivial collaboration deserves a new piece of documentation.

We must have all experienced the frustration of searching for a small piece of useful text through a huge body of documentation. Without a good organization of the documentation, there are just too many irrelevant pages. One implication of the different use frequency of features suggests that documentation be ordered in such a way that the most relevant parts appear first. Moreover, tools that can speed the efficiency of retrieving documentation can also be useful.

3.2. Feature Prioritizing and Default Implementation

Of course designers should always try to minimize the learning curve, for example, by appealing to well-known design convention like design patterns. We have not seen much literature devoted to this topic. In this subsection one example is presented, which is called feature prioritizing and default implementation.

Default implementation is a design technique that can effectively smoothen the learning curve. For a complex hot spot like the tree model of the JTree component, the task of understanding the TreeModel interface and implementing an appropriate class for it seems just too daunting for all but the most experienced one to tackle. Alternatively, designers can first prioritize all the possible solutions for a hot spot and then choose the one that is most likely to be used as the default. By doing so the programmers do not have to deal with too many details at a time. As the programmers use the framework longer, incrementally, they will be able to learn more and more about the hot spot. Eventually, they will be able to replace the default implementation by their own’s. Essentially, this technique works by distributing the learning curve over a longer period of time.

The implementation for the JTree component provides default implementations for both TreeModel and TreeNode, and three other interfaces. This has made learn the component much easier. We started with using the component without even knowing the details of the interfaces, and we found that in many cases, it is unnecessary to know these details. When harder requirements with regard

to the use of the component are encountered, chances are that programmers have already gained enough knowledge about the component so that they are able to figure out the more complex customization by themselves, which would be too complex for them to handle previously.

3.3. 'Cognitive' Patterns

We define 'cognitive' patterns as typical designs that are frequently used but difficult for average programmers to understand. We claim that having a good knowledge of cognitive patterns can help to decrease the learning curve and avoiding common errors. In contrast to design patterns, which tend to be general, a cognitive pattern can be specific to a particular framework. Several examples are as follows:

1. Hidden global objects

Hidden global objects are singletons that are implicitly initialized by a framework. The focus here is on 'hidden', 'hidden' makes easy things hard. Experience reveals that it is usually difficult to dig out these objects when needed. One example is the `ToolTipManager` that we introduce before.

2. Context-wrapping objects

Some objects actually wrap the context of a collaboration. To understand such objects, we have to identify the context for the collaboration to understand them as a whole. Without knowing this would make learning a framework more painstaking. The `TreeCellRenderer` object is such an example.

3. Superclass-maintained invariants

Sometimes a method of the super class can implicitly do something that will affect the behavior of subclass method. One such good example is MFC's message handlers. In base class `CWnd`, there is a default message handler that provides default behavior for each type of message. One side effect of this method is to save a copy of its parameter event. Users can provide their own message handlers to process events in their own way but they are warned not to expect to change the event and pass it back to the framework. This warning appears repeatedly and identically in the documentation of every message handler. A lot of programmers complained that the documentation is confusing.

4. Use of new language features

It is usually hard to understand functionalities that are implemented by unfamiliar language features. For example, Swing uses Java's reflection mechanism to implement PLAF (Pluggable Look And Feel) which would have been implemented as an abstract factory in C++.

4. Summary and Future Work

Object-Oriented frameworks require tremendous effort from the part of the users. One change from traditional application engineering is that program understanding now becomes a first class activity. Based on experience with and observation on how programmers use GUI frameworks, we discuss aspects that can have impact on programmer's performance and provide some advices that may improve the state of practice. We informally refer to the '7+/-2' principle in our discussion.

The object-oriented framework approach also poses some new research challenges. For instance, new language constructs may be needed to better support the structuring and composing of frameworks; research such as mixin layer, AOP [1] and [10] are examples. We suspect that if we

view the many questions that programmers asked as an information retrieval problem, then good question and answering tools may be useful in helping those who work in the trench.

In the future, we plan to collect more evidences to both consolidate and refine our understanding of the issue. Some of the points may need experiments to validate. We are also interested in the feasibility of measuring the complexity of hot spots, which would be useful to predict the potential problematic areas of a framework.

References

- [1] ACM. *Special Issue on Aspect-Oriented Programming, Communications of ACM*. October 2001.
- [2] D. Batory, R. Cardone, and Y. Smaragdakis. Object oriented frameworks and product lines. In *Proceedings of the First Software Product Line Conference*, Denver, Colorado, August 2000.
- [3] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA 1989*, October 1989.
- [4] G. Froehlich, H. J. Hoover, L. Liu, and P. G. Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 1997 International Conference on Software Engineering*, Boston, Mass., May 1997.
- [5] E. Gamma, R. Helm, R. E. Johnson, and J. O. Vlissides. *Design Patterns-Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [6] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of ECOOP/OOPSLA 90*, Ottawa, Canada, 1990.
- [7] R. E. Johnson. Documenting frameworks with patterns. In *Proceedings of OOPSLA 92*, Vancouver, Canada, 1992.
- [8] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(12):22–25, 1988.
- [9] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3), August-September 1988.
- [10] D. Manolescu and A. Kunzle. Why java is not suitable for object-oriented frameworks. In *Companion to the Proceedings of OOPSLA 2001*, Tampa bay, Florida, USA, October 2001.
- [11] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.
- [12] D. Roberts and R. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of PLoP'96*, 1996.