# Evolving Frameworks

A Pattern Language for Developing Object-Oriented Frameworks

Don Roberts, Ralph Johnson, University of Illinois {droberts,johnson}@cs.uiuc.edu

## Introduction

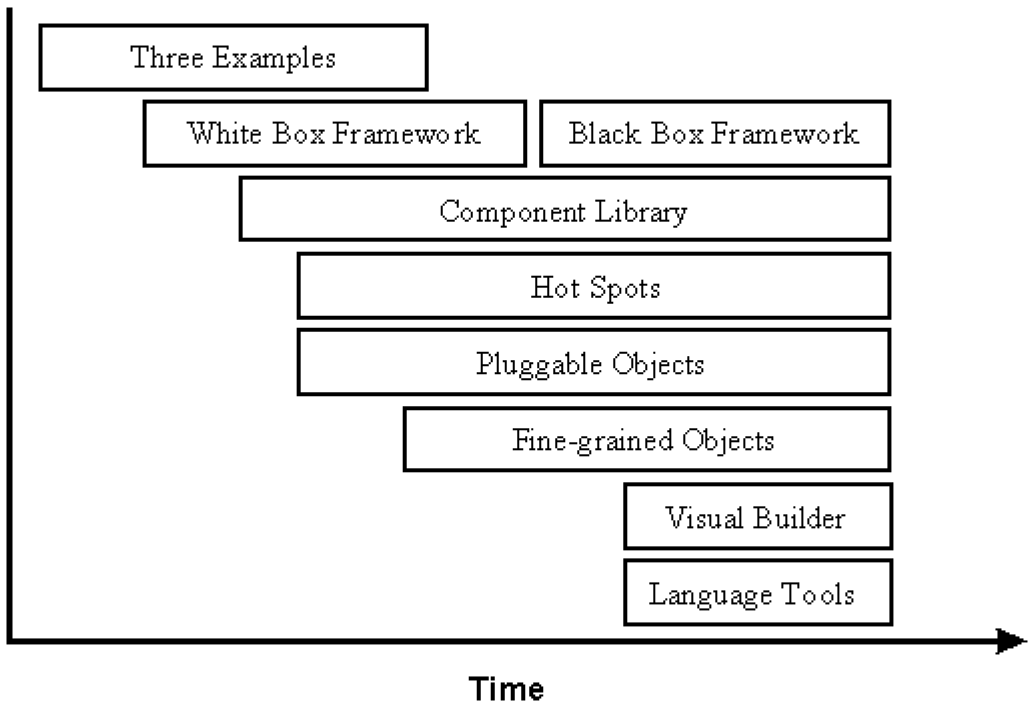
Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate. A good framework can reduce the cost of developing an application by an order of magnitude because it lets you reuse both design and code. They do not require new technology, because they can be implemented with existing object-oriented programming languages.

Unfortunately, developing a good framework is expensive. A framework must be simple enough to be learned, yet must provide enough features that it can be used quickly and hooks for the features that are likely to change. It must embody a theory of the problem domain, and is always the result of domain analysis, whether explicit and formal, or hidden and informal.

Therefore, frameworks should be developed only when many applications are going to be developed within a specific problem domain, allowing the time savings of reuse to recoup the time invested to develop them.

The patterns in this paper form a pattern language. A pattern language is a set of patterns that are used together to solve a problem. In general, you start at the beginning of a pattern language and work to the end. However, many of these patterns are applied over and over again, and so you will be working on many of them at once.

Figure 1 shows how the patterns are related to each other in a sort of time-line. Some patterns never overlap. Others usually do. When two patterns have the same time-span, it means that sometimes one goes first, sometimes the other. The "rationale" and "implementation" sections of the patterns often discuss the complexities that are hidden by a diagram like Figure 1.



Evolving Frameworks describes a common path that frameworks take, but it is not necessary to follow the path to the end to have a viable framework. In fact, most frameworks stop evolving before they reach the end. In some cases, this is because the frameworks die; they aren't used any more and so don't change any more. In other cases, it is because it is better for the frameworks to stay more white-box. We hope that the forces for the various patterns describe why.

## Three Examples

### Context

You have decided to develop a framework for a particular problem domain.

### Problem

How do you start designing a framework?

### Forces

- People develop abstractions by generalizing from concrete examples. Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart. A framework is a reusable design, so you develop it by looking at the things it is supposed to be a design of. The more examples you look at, the more general your framework will be.
- Designing applications is hard. You can't have too many examples or you'll never get your framework done.
- Having a framework makes it easier to develop applications, even if it is only marginally useful. Once you get the first version of the framework, it will be easier to develop more examples.
- Projects that take a long time to deliver anything tend to get canceled. There is usually a window of market opportunity that must be met. If the project doesn't start making money sometime, the organization will run into cash-flow problems. Political fortunes within a company rise and fall, and a system must be built before its champions move on to other interests or other companies.

### Solution

Develop three applications that you believe that the framework should help you build. These applications should pay for their own development.

### Rationale

Developing reusable frameworks cannot occur by simply setting down and thinking about the problem domain. No one has the insight to come up with the proper abstractions. Domain experts won’t understand how to codify the abstractions that they have in their heads, and programmers won’t understand the domain well enough to derive the abstractions. In fact, often there are abstractions that do not become apparent until a framework has been reused. Quite often these "hidden" abstractions have no physical analog (e.g., Strategy objects).

While initial designs may be acceptable for single applications, the ability to generalize for many applications can only come by actually building the applications and determining which abstractions are being reused across the applications. Generalizing from a single application rarely happens. It is much easier to generalize from two applications, but it is still difficult. The general rule is: build an application, build a second application that is slightly different from the first, and finally build a third application that is even more different than the first two. Provided that all of the applications fall within the problem domain, common abstractions will become apparent.

Your framework won't be done after three applications. You can expect it to continue to evolve. However, it should be useful and you can use it to gather more examples. Just don't acquire too many users initially the framework *will* change!

There are two approaches to developing these applications. In the first approach, the applications are developed in sequence by a single team. This allows the team to begin reusing design insight immediately at the possible expense of narrowness. In the second approach, the applications are developed in parallel by separate teams. This approach allows for diversity and different points of view at the expense of the time it will take to unify these applications in the future.

Some people have built a particular kind of application many times, so they might be able to design a framework without first building an example. They are not counterexamples, they just built their examples before they decided to start the framework.

### Implementation

The most obvious way to follow this pattern is to simply build three applications in succession, making sure that both code and people are carried over from one project to the next. But sometimes people build a series of applications without trying to reuse any code, and then suddenly realize they should build a framework. Since they've already developed their three applications, they can often do a good job at the framework the first time.

Another way to follow this pattern is to prototype several applications without building industrial strength versions of any of them. You'll have to refactor it when you use it, but you'll be a lot closer than you would be after one application. An advantage of this approach is that you can tell your customer that they are only buying the rights to use the framework, not complete ownership of it. Even though the application will force you to change the framework, you will still retain ownership of it. When you build a series of applications, it is often hard to get the right to use code written for one to build the next.

Note that no special object-oriented design techniques are needed when you are building these applications. Just use standard techniques, and try to make your system flexible and extensible.

### Examples

The Runtime System Expert framework was initially developed by developing runtime systems for various platforms. The first platform was Tektronix Smalltalk. The second platform was ParcPlace Smalltalk. [Durham, Johnson, 1996] Bill Reynen created a C front-end for the RTL system which required a C runtime system (which was quite trivial)

We don’t know what the original examples were that the designers of MVC had in mind. We do know that the usage of MVC in many applications has affected how it evolved from its roots into the framework currently implemented in VisualWorks 2.5.

### Related Patterns

This is a special example of the Rule of Three [Tracz 1988]. The initial versions of the framework will probably be white-box frameworks.

---

## White-box Framework

### Context

You have started to build your second application.

### Problem

Some frameworks rely heavily on inheritance, others on polymorphic composition. Which should you use?

### Forces

- Inheritance results in strong coupling between components, but it lets you modify the components that you are reusing, so you can change things that the original designer never

imagined you would change.
- Making a new class involves programming.
- Polymorphic composition requires knowing what is going to change
- Composition is a powerful reuse technique, but it is difficult to understand by examining static program text.
- Compositions can be changed at runtime.
- Inheritance is static and cannot be easily changed at runtime.

### Solution

Use inheritance. Build a white box framework [Johnson, Foote, 1988] by generalizing from the classes in the individual applications. Use patterns like Template Method and Factory Method [Gamma et al., 1995] to increase the amount of reusable code in the superclasses from which you are inheriting. Don't worry if the applications don't share any concrete classes, though they probably will.

### Rationale

Inheritance is the most expedient way of allowing users to change code in an object-oriented environment since it is supported by most object-oriented languages. This allows you to create new classes by simply inheriting most of the desired behavior from an existing class and overriding only the methods that are different in the subclass. At this point, you should not be worrying about the semantics of inheritance, just the ability to reuse existing code. Once you have a working framework, you can start using it. This will show you what is likely to change and what is not. Later, immutable code can be encapsulated and parameterized by converting the framework into a black-box framework. However, at this point in the life cycle, you probably do not have enough information to make an informed decision as to which parts of the framework will consistently change across applications and which parts will remain constant.

### Implementation

While developing the subsequent applications, whenever you find that you need a class that is similar to a class that you developed in a prior application, create a subclass and override the methods that are different. This is known as *programming-by-difference* [Johnson, Foote, 1988]. After you've made a couple of subclasses, you will recognize which parts you are consistently overriding and which parts are relatively stable. At that point, you will be able to create an abstract class to contain the common portions.

Also, while overriding methods, you will probably discover that certain methods are almost the same in all of the subclasses. Again, you should factor out the parts that change into a new method. By doing this, the original methods will all become identical and can be moved into the abstract class.

### Examples

The Model-View-Controller framework for graphical user interfaces was originally a white-box framework. New view and controller classes were created by making subclasses of the **View** and **Controller** classes, respectively. For instance, to create a scrolling view, a programmer would have to create a new subclass of **ScrollController** to handle the scrolling behavior for the view.

### Related Patterns

As you develop additional applications, you should begin to build a component library. Black-box framework addresses the same problem, but in a different context.

---

## Component Library

### Context

You are developing the second and subsequent examples based on the white-box framework.

### Problem

Similar objects must be implemented for each problem the framework solves. How do you avoid writing similar objects for each instantiation of the framework?

### Forces

- Bare-bones frameworks require a lot of effort to use. Things that work out-of-the-box are much easier to use [Foote, Yoder, 1996]. A framework with a good library of concrete components will be easier to use than one with a small library.
- Up front, it is difficult to tell which components users of the framework will need. Some components are problem-specific while others occur in most solutions.

### Solution

Start with a simple library of the obvious objects and add additional objects as you need them.

### Rationale

As you add objects to the library, some will be problem-specific and never get reused. These will eventually be removed from the library. However, these objects will provide valuable insight into the type of code that users of the framework must write. Others will be common across most or all solutions. From these, you will be able to derive the major abstractions within the problem domain that should be represented as objects in the framework.

### Implementation

These objects are the concrete subclasses of the abstract classes that make up the framework. Abstract classes generally lie in the framework. Concrete classes generally lie in applications. The component library can be created by accumulating all of the concrete classes created for the various applications derived from the framework. In the long run, a class should only be included in the component library if it used by several applications, but in the beginning, you should put all of them in. If a component gets used a lot, it should remain in the library. If it never gets reused, it gets throw out. Many components will get refactored into smaller subcomponents by later patterns and disappear that way.

### Examples

The first step in creating a component library for MVC, which took place in the early 80's, was the creation of *pluggable views*. Pluggable views provided a way to adapt a controller and view pair to the interface of a particular model. For example, **SelectionInListView** was a pluggable view that provided a list view that could be adapted to any model that contained a list. Despite this step, MVC as a whole was still fairly white-box, since you had to make a new subclass to do anything other than adapt the view to a particular model.

### Related Patterns

As components get added to the library, you will begin to see recurring code that sets of components share. You should look for hot spots in your framework where the code seems to change from application to application.

---

## Hot Spots

### Context

You are adding components to your component library.

## Problem

As you develop applications based on your framework, you will see similar code being written over and over again. Wolfgang Pree calls these locations in the framework "hot spots." [Pree 1994] How do you eliminate this common code?

## Forces

- If changeable code is scattered across an application, it is difficult to track down and change.
- If changeable code is located in a common location, program flow can be obfuscated since calls are always being made to the object that contains the changeable code.

## Solution

Separate code that changes from the code that doesn't. Ideally, the varying code should be encapsulated within objects whenever possible, since objects are easier to reuse than individual methods. With the code encapsulated, variation is achieved by composing the desired objects rather than creating subclasses and writing methods.

## Rationale

If the framework is being reused extensively (as it should be), certain pieces will vary often. By gathering the code that varies into a single location (object) it will both simplify the reuse process *and show users where the designers expect the framework to change.* Good names for these objects will make the control flow less important to understanding the framework [Beck].

## Implementation

Many of the Gang of Four design patterns encapsulate various types of changes. The following table shows possible design patterns to use when different portions of the framework change from application to application: [Gamma et al., 1995]

| **What varies** | **Design Pattern** |
|---|---|
| Algorithms | Strategy, Visitor |
| Actions | Command |
| Implementations | Bridge |
| Response to change | Observer |
| Interactions between objects | Mediator |
| Object being created | Factory Method, Abstract Factory, Prototype |
| Structure being created | Builder |
| Traversal Algorithm | Iterator |
| Object interfaces | Adapter |
| Object behavior | Decorator, State |

## Examples

Objectworks 4.0 began to make extensive use of wrappers, which are an example of the Decorator pattern. The code to add features to **View**s was stored in the wrappers. This was in response to users of the framework having to write similar code every time they wanted to add a particular function to a view or controller. In the new version of the framework, window features are added by adding the appropriate wrapper for each desired behavior. For instance, to create a view with a menu bar and scrolling behavior, the programmer simply adds a **ScrollWrapper** and a **MenuBarWrapper** to his view.

### Related Patterns

To encapsulate the hot spots, you will often have to create finer grained objects. Often these fine-grained objects will cause your framework to become more black-box.

---

## Pluggable Objects

### Context

You are adding components to your component library.

### Problem

Most of the subclasses that you write differ in trivial ways. (e.g., Only one method is overridden) How do you avoid having to create trivial subclasses each time you want to use the framework?

### Forces

- New classes, no matter how trivial, increase the complexity of the system.
- Complex sets of parameters make parameterized classes more difficult to understand and use.

### Solution

Design adaptable subclasses that can be parameterized with messages to send, indexes to access, blocks to evaluate, or whatever else distinguishes one trivial subclass from another.

### Rationale

If the differences between subclasses is trivial, creating a new subclass just to encapsulate the small change is overkill. Adding parameters to the instance creation protocol provides for reuse of the original class without resorting to programming.

### Implementation

Determine what changes in each of the subclasses that you have been required to write. If the difference is simply in some constant, symbol, or class reference, create an instance variable to contain the reference and pass it into the object in the instance creation method. If the variation is in a small piece of code, pass a block representing the code to the instance creation method and, again, store it in an instance variable. If you are in an environment that does not support blocks, such as C++, use the available facilities for anonymous functions, such as function pointers.

### Examples

Early versions of the Model-View-Controller framework included the notion of pluggable views. These were standardized view classes (e.g., **SelectionInListView**) that could be opened on any model provided that the appropriate messages to retrieve the information from the model were passed as parameters when the view was created. In more recent versions of the framework, the pluggable views have been mostly supplanted by **PluggableAdaptors** that allow the standard **value** and **value:** messages that **Models** must understand to be translated into arbitrary pieces of code stored in blocks. This essentially allows **Views** to be plugged into any object regardless of what set of messages it understands.

### Related Patterns

Creating pluggable objects is one way to encapsulate the hot spots in your framework. The parameters can be automatically supplied by a visual builder.

---

## Fine-grained Objects

### Context

You are refactoring your component library to make it more reusable.

### Problem

How far should you go in dividing objects into smaller objects?

### Forces

- The more objects there are in the system, the more difficult it is to understand.
- Applications can be created by simply choosing the objects that implement the functionality that is desired within the application. No programming is required.

### Solution

Continue breaking objects into finer and finer granularities until it doesn't make sense to do so any further. That is, dividing the object further would result in objects that have no individual meaning in the problem domain.

### Rationale

Since frameworks will ultimately be used by domain experts (non-programmers) you will be providing tools to create the compositions automatically. Therefore, it is more important to avoid programming. The tools can be designed to manage the proliferation of objects.

### Implementation

Anywhere in your component library that you find classes that encapsulate multiple behaviors that could possibly vary independently, create multiple classes to encapsulate each behavior. Wherever the original class was used, replace it with a composition that recreates the desired behavior. This will reduce code duplication, as well as the need to create new subclasses for each new application.

### Examples

Beginning with Objectworks 4.0, the Model-View-Controller became more fine-grained. This was accomplished by using new objects that represented finer grained concepts than the original **Model**, **View**, and **Controller** classes. A couple of examples of these objects were **Wrappers**, which allowed a unit of functionality to be added to any view, and **ValueHolders**,[Woolfe 1994] which allowed views to depend only on a portion of a model rather than the entire model. With this version, to make a view scrollable, the programmer only needed to apply a **ScrollWrapper** to his view.

### Related Patterns

As the objects become more fine-grained, the framework will become more black-box.

---

## Black-box Framework

### Context

You are developing pluggable objects by encapsulating hot spots and making fine-grained objects.

## Problem

Some frameworks rely heavily on inheritance, others on polymorphic composition. Which should you use?

## Forces

- Inheritance results in strong coupling between components, but it lets you modify the components that you are reusing, so you can change things that the original designer never imagined you would change.
- Making a new class involves programming.
- Polymorphic composition requires knowing what is going to change
- Composition is a powerful reuse technique, but it is difficult to understand by examining static program text.
- Compositions can be changed at runtime.
- Inheritance is static and cannot be easily changed at runtime.

## Solution

Use inheritance to organize your component library and composition to combine the components into applications. Essentially, inheritance will provide a taxonomy of parts to ease browsing and composition will allow for maximum flexibility in application development. When it isn't clear which is the better technique for a given component, favor composition.

## Rationale

A black-box framework is one where you can reuse components by plugging them together and not worrying about how they accomplish their individual tasks [Johnson, Foote, 1988]. In contrast, white-box frameworks require an understanding of how the classes work so that correct subclasses can be developed.

People like to organize things into hierarchies. These hierarchies allow us to classify things and quickly see how the various classifications are related. By using inheritance, which represents an is-a relationship, to organize our component library, we can rapidly see how the myriad of components in the library are related to each other. By using composition to create applications, we both avoid programming and allow the compositions to vary at runtime.

## Implementation

Convert inheritance relationships to component relationships. Pull out common code in unrelated (by inheritance) classes and encapsulate it in new components. Many of the previous patterns will provide the techniques for locating and creating new component classes.

## Examples

VisualWorks is MVC turned into a black-box framework. Now, rather than creating various subclasses of **View**, or even reusing various subclasses of view, we take a generic view and add wrappers to it corresponding to the various behaviors that we require. In the same way, rather than creating a complex model class and ensuring that dependencies get updated correctly, we simply compose a bunch of **ValueHolders** to hold the values in our model and let them worry about updating their dependents.

Another example of this is Alan Durham's Runtime System Expert for the Typed Smalltalk Compiler.[Durhan, Johnson 1996]

## Related Patterns

By organizing the component library in the manner, we support the creation of a visual builder that allows the library to be browsed and compositions to be created graphically.

---

## Visual Builder

### Context

Your now have a black-box framework. You can now make an application entirely by connecting objects of existing classes. The behavior of your application is now determined entirely by how these objects are interconnected. A single application consists of two parts. The first part is the script that connects the objects of the framework together and then "turns them on." The second part is the behavior of the individual objects. The framework provides most of the second part, but the application programmer must still provide the first part.

### Problem

The connection script is usually very similar from application to application with only the specific objects being different. How do you simplify the creation of these scripts?

### Forces

- The compositions that represent applications of the framework are convoluted and difficult to understand and generate.
- Building tools is expensive.
- Domain experts are rarely programmers.

### Solution

Make a graphical program that lets you specify the objects that will be in your application and how they are interconnected. It should generate the code for an application from its specification.

### Rationale

Since the code is basically just a script, the tool can generate it automatically. The tool will also make the framework more user-friendly by providing a graphical interface to it that should draw on the standard notations present in the problem domain. At this point, domain experts can create applications by simply manipulating images on the screen. Only in rare cases should new classes have to be added to the framework.

### Implementation

Sometimes you can specify the components and relationships in an application entirely with dialog boxes and browsers. But usually, you need to draw graphs to represent the complex relationships present in more complicated domains. In this case, you should use a framework for graphical editors like HotDraw.[Johnson 92]

### Examples

In Visualworks 1.0, ParcPlace provided an interface builder to allow users to "paint" the GUI on a canvas. The builder takes the graphical description of the GUI and creates the application from it. It does this by creating an **ApplicationModel** with **ValueHolders** of the appropriate type for each control. It creates composite **View** objects that make up the main View and adds Wrappers to create the appropriate functionality for each widget. All of this information is stored in a windowSpec, which is a declarative description of the entire GUI. At runtime, the windowSpec is fed to a **UIBuilder** that interprets it, creates the objects it describes, and composes them to create the final user-interface. **UIBuilder** is implemented as an interpreter [Gamma et al., 1995].

### Related Patterns

Congratulations! You have just developed a visual programming language. Note that this implies that you will need language tools, just like any other language.

---

## Language Tools

### Context

You have just created a builder.

### Problem

The visual builder creates complex composite objects. How do you easily inspect and debug these compositions?

### Forces

- Existing tools are usually inadequate for dealing with the specialized composition relationships present in the framework.
- Building good tools is an expensive task that can be viewed as overhead.

### Solution

Create specialized inspecting and debugging tools.

### Rationale

Since the system you have created is essentially a graphical, domain-specific, programming language, it will require language tools to help debug and understand it. The tools that came with the language that you built your framework in will probably not be as good as they should be, because your framework will be filled with little objects that all look alike, and half of them will be completely uninteresting to someone who just wants to build an application.

### Implementation

Find the portions of your framework that are difficult to inspect and debug. These can usually be found where objects are being composed extensively using things such as wrappers and strategies. Create specialized tools to navigate and inspect the compositions. These tools should allow the user to elide portions of the composition that are not interesting.

### Examples

Visualworks doesn't have this yet, but it is the next logical step in its evolution. Many of the long-time MVC programmers have complained that the current framework is more difficult to use since the views are large hierarchical structures of composed wrappers on wrappers. The real problem is that there are no inspectors or debuggers specifically designed for handling these compositions. If such tools existed, debugging or inspecting a view should be no more difficult than inspecting any other object in the system.

---

### Acknowledgments

We would like to thank Ward Cunningham for his shepherding and valuable insight that is reflected in the final version of this paper. We would also like to thank the UIUC patterns group for their scathing reviews that helped us discover some of the weaknesses in our presentation.

### References

Beck K. *Smalltalk Best Practice Patterns — Volume 1: Coding*. Prentice-Hall, 1996.

Durham A, Johnson R. A Framework for Run-time Systems and its Visual Programming Language. In *Proceedings of OOPSLA ’96, Object-Oriented Programming Systems, Languages, and Applications*. San Jose, CA, October 1996.

Foote B, Opdyke W. Life Cycle and Refactoring Patterns that Support Evolution and Reuse. *First Conference on Pattern Languages of Programs (PLoP’94)*. Monticello, Illinois, August, 1994. *Pattern Languages of Program Design*. Edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995.

Foote B, Yoder J. Attracting Reuse. *Third Conference on Pattern Languages of Programs (PLoP'96)*, Monticello, Illinois, September 1996.

Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley, 1995.

Johnson R, Foote B. Designing Reusable classes. *Journal of Object-Oriented Programming*, 1 (2):22–35, June/July 1988.

Pree W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading Mass., 1994.

Tracz W. RMISE Workshop on Software Reuse Meeting Summary. In *Software Reuse: Emerging Technology*, pages 41-53. IEEE Computer Society Press, Los Alamitos, CA 1988.

Durham A, Johnson R. A Framework for Run-time Systems and its Visual Programming Language. In *Proceedings of OOPSLA ’96, Object-Oriented Programming Systems, Languages, and Applications.* San Jose, CA. October 1996.

Woolfe B. Understand and Using ValueModels. *http://www.ksccary.com/valujrnl.htm*. 1994.