Building Reusable Components for Real-Time Imaging Systems

Raghvinder S. Sangwan, Robert S. Ludwig, Colin J. Neill, and Phillip A. Laplante▲†

Software Engineering Group, Engineering Division, Pennsylvania State University, Great Valley Graduate Center, Malvern, Pennsylvania, USA

Imaging systems are traditionally developed using structured analysis and design techniques. While there are many reasons that engineers choose this approach, one is the expected real-time performance benefits. But structured approaches tend to be rigid with respect to changing needs, technologies, devices, and algorithms. More generally, these systems are difficult or impossible to reuse because each new problem requires a new solution. Object-oriented approaches, on the other hand, can lead to systems that are more readily reused if certain best practices are followed. However, the conventional wisdom is that the price for such benefits is degraded real-time performance.[1] The contribution of this work is an examination of these best practices, in the form of patterns and design principles, with reference to imaging systems. Then an extensive implementation of these practices is done on an existing imaging system, Kahindu, which is a teaching package built using the object-oriented paradigm. We then show how by applying these best practices not only improved structure is obtained, but surprisingly, improved performance as well. Our results challenge the conventional belief that the "price" for the improved structure, ease-of-extension, maintainability, etc. of object-oriented systems in imaging systems is degraded performance.

Journal of Imaging Science and Technology 49: 154–162 (2005)

An Imaging System and Its Current Design

Imaging systems, like many other scientific and engineering systems, incorporate complex software systems. These systems are typically designed and built with algorithmic correctness as the primary, and often only, goal. In real-time imaging systems this design goal is extended to include timeliness and possibly determinism, but it is very unusual for any other quality criteria to be considered, such as robustness, maintainability or reusability, and this is a detriment to the resulting systems. Neill and Laplante have studied this phenomenon and proposed the use of contemporary software engineering practices and methodologies in the development of such imaging systems.[1–3] This work puts those ideas into practice by reengineering an existing imaging system, Kahindu, to make it more maintainable, reusable and generally simpler to understand and extend.

Kahindu is an image processing system written in the Java programming language to be used in conjunction with the first book on image processing in Java.[4] The program supports demonstration of many of the image processing algorithms described in the book such as radix-2 FFTs, the Prime Factor Algorithm, convolution, filtering, warping, and many more. While a Web based second edition of the Kahindu program is available, we chose to use the first edition because the source code was available with the book. In fact, we don't intend for our work to be a criticism of this tool, which was designed for a specific teaching purpose and not industrial strength applications. We chose to use Kahindu as a basis for our experiments because we believe it to be representative of much of the commercial image processing software available.

In order to analyze the structural characteristics of Kahindu we used Headway reView, a software visualization and analysis tool used for code comprehension.[5] This tool creates hierarchical directed graphs (or higraphs) to visualize the structure of a software system. In the highest level view, it shows the system's components and subsystems and their relationships. A user can then drill down into each component uncovering details of the classes that make up a component and their interrelationships. At the lowest, class level view, the methods and attributes of each class are visible. One can, therefore, use reView to assess the quality of a software system using static software quality metrics as described in various works.[6–9]

Figure 1 shows the first snapshot of Kahindu created by reView. There are seven packages representing the different subsystems in Kahindu. The `Main` class is not part of any subsystem.

Instead of showing all the arcs for dependencies between any two packages, reView collapses them into a single arc and labels it with the number of dependencies. For example, The `gui` package has 30 dependencies on the vs package and the vs package has four dependencies on the `gui` package.

Original manuscript received July 14, 2004

▲ IS&T Member

†Corresponding Author: P. Laplante, plaplante@gv.psu.edu

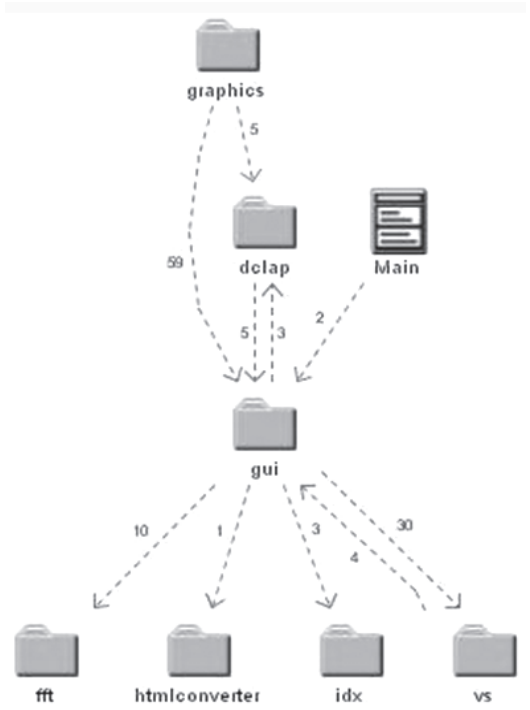

Figure 1. Package dependency higraph for Kahindu. Nodes represent packages, edges represent dependencies.

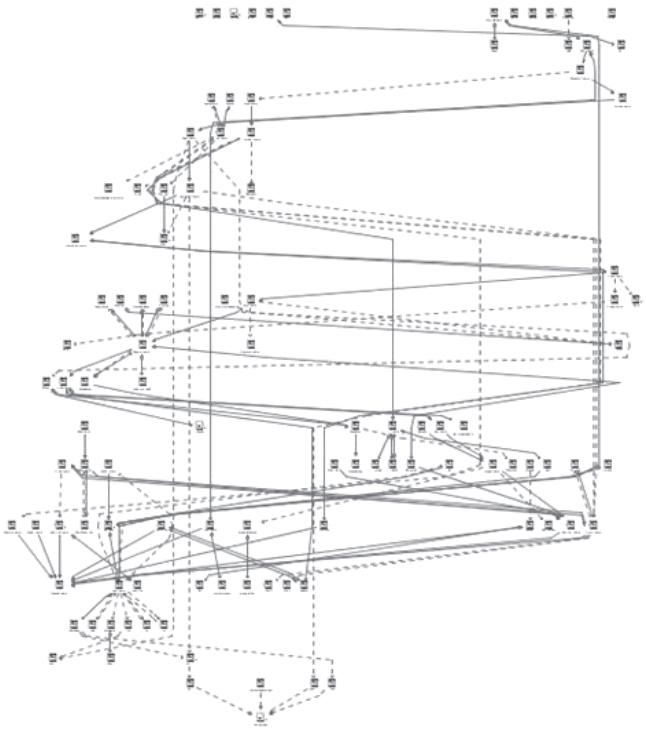

Figure 2. Class coupling higraph for the `gui` package. Nodes represent classes, edges represent dependencies.

From this view it is not particularly obvious that the design suffers any significant flaws, but on inspection the package dependency higraph shows the violation of the acyclic dependency principle[8] which advocates there be no cycles in a package dependency graph. In the current design, there exists a cycle between the `gui` and vs packages, and between `gui` and `dclap` packages. By introducing interdependence among packages, cycles make the design of a system rigid, fragile and difficult to reuse. In a rigid system a single change within heavily interdependent packages can initiate a cascade of changes in dependent packages. A single change in such software also has the potential of causing problems in parts that bear no conceptual relationship with the piece that was changed, thus making a system fragile. It is subsequently difficult to reuse individual components when those desirable parts of the system are tangled with parts that are not desired.

Continuing the analysis by drilling down into each package revealed `gui` to be the most complex. Figure 2 is a snapshot of this package.

Due to the limited space available, one cannot read the class names and get a sense of the number of dependencies from this diagram (in the tool one can zoom in on the area of interest). However, there are 107 classes with 5190 connections among them and despite the limitations of the paper medium; the diagram does give one a sense of this complexity inherent in the `gui` package. Indeed, analysis reveals this package to be highly unstable. Instability is the ratio between efferent coupling and total coupling.[8] Total coupling is the sum of afferent (incoming) and efferent coupling (out-going). Afferent coupling indicates the responsibility of a package and is measured by how many external packages are dependent upon it. Efferent coupling indicates the independence of a package and is measured by how many packages it is dependent upon. By examining the directed arcs into and out of the `gui` package from Fig. 1 we find three packages that are dependent on `gui` and five packages that `gui` is dependent on. The instability ratio, therefore, is 5/8 = 0.625 implying that it is not very resilient to change. That is, any changes to the packages it is dependent upon are likely to cause it to change as well.

Another measure of design complexity is the average response factor for the classes in a package.[7] The response factor measures the set of all methods that can be invoked in response to a single message received by an object of a given class. The higher the response factor, the more difficult it is to understand, and therefore, debug and test such a class. The average response factor for the classes in the `gui` package is 15.5 with the worst case being 103. A single message received by an object of a class in the `gui` package, therefore, leads to the calling of approximately 16 other methods on an average and 103 other methods in the worst case.

The depth of inheritance hierarchies[7] among classes in this package is also very high which increases the complexity of the package. While inheritance is often considered a benefit of object-oriented systems in this case it has been used merely as a code-sharing mechanism and those benefits are lost against the increased code dependencies and general reduction in clarity. Figure 3 shows the inheritance hierarchy for the `TopFrame` class. This turned out to be the central class to the Kahindu system; the class from which most of the useful classes in the system are derived. In this view the inheritance tree is inverted with the `TopFrame` class at the bottom. The depth of this inheritance hierarchy is 19, meaning that there are 19 levels of subclassing from the topmost class, `TopFrame`, to the lowest-level classes. This implies the classes forming the leaves of

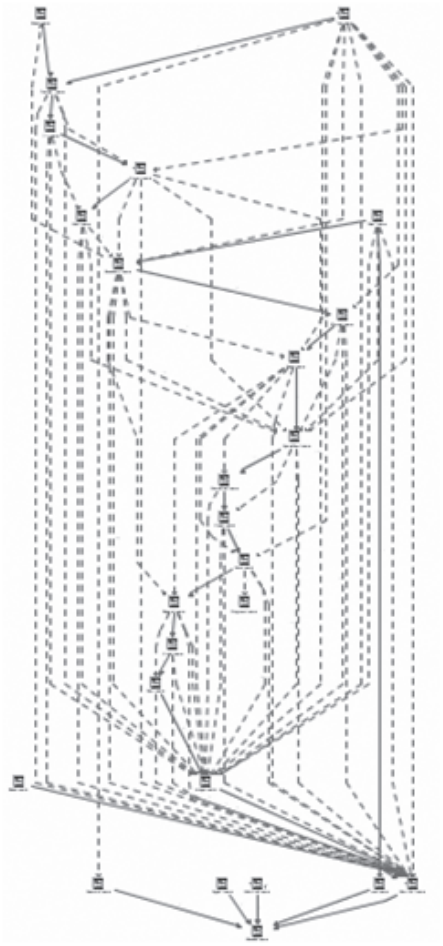

Figure 3. Inheritance hierarchy for the `TopFrame` class. Nodes represent classes, edges represent the inheritance chain.

this hierarchy inherit a lot of methods from their ancestral classes making their design and behavior very complex, and therefore, harder to understand and test. As a rule of thumb, an inheritance hierarchy should not get deeper than five to eight levels of specialization.

In addition the package also suffers from several other negative characteristics known as code smells.[10] It has a large number (25) of data classes–classes that contain primarily data and no interesting methods; eight classes with feature envy methods–methods that are more interested in the data of classes other than their own thus breaking the object-oriented principle where a class is an encapsulation of data and methods that act on that data (and not the data of another class); and many large classes with up to 75 methods, far too many responsibilities folded into a single class.

Patterns

Overall then it is clear that the original Kahindu design was far from ideal, despite the fact that it works. This underscores the well known fact that while developing any software is hard, developing reusable software is even harder.[11] Most designs solve a specific problem, but they should also be general enough to address future problems and requirements. In image processing, this reality manifests as a way to reuse existing designs by finding recurring patterns and using them as a basis for new designs. But how can this be done without adversely impacting performance because of the necessary abstractions introduced?

Fortunately, when properly employed, object-oriented design has the capability to include distinct elements that can cater to future changes and extensions of the designed system. These "design patterns" were first introduced to the mainstream of software engineering practice by Gamma et al.[11] and are commonly known as the "Gang of Four (GoF)" patterns. While many more sets of patterns have since been published we have concentrated on the GoF patterns in this study.

The application of patterns to the design of image processing systems was first discussed in Neill and Laplante[3] and for the purposes of this work we only describe a small subset of the GoF patterns needed for our purposes.

First, consider the Strategy pattern. It is intended to define a family of reusable algorithms that are easily interchangeable for one another during runtime. Because the interfaces to the image objects, filters, and display algorithms should be similar, such an approach makes sense. The generic structure of the Strategy pattern is depicted using a UML class diagram in Fig. 4.

The second pattern that is used in our reengineered system is the Decorator pattern. This pattern is used to attach additional functionality (to "decorate") an object after it has been created (Fig. 5). Decorators allow for the flexible and transparent addition or deletion of responsibilities (functionality) from an object at runtime. Moreover, decorators embody the notion that composition is preferable to inheritance as it avoids the fixed overhead and enables easier reuse.

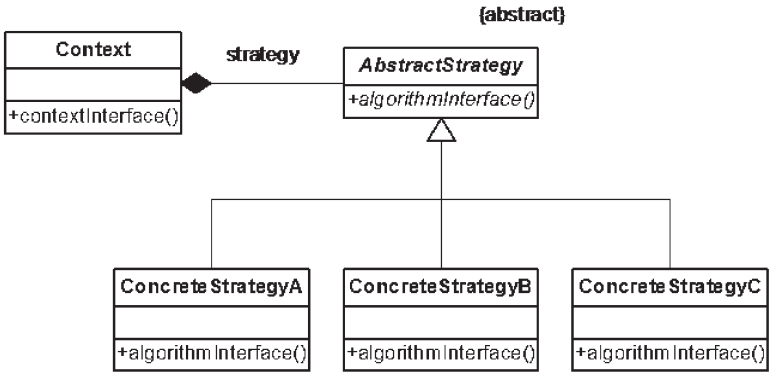

Figure 4. The structure of the strategy pattern.[11]

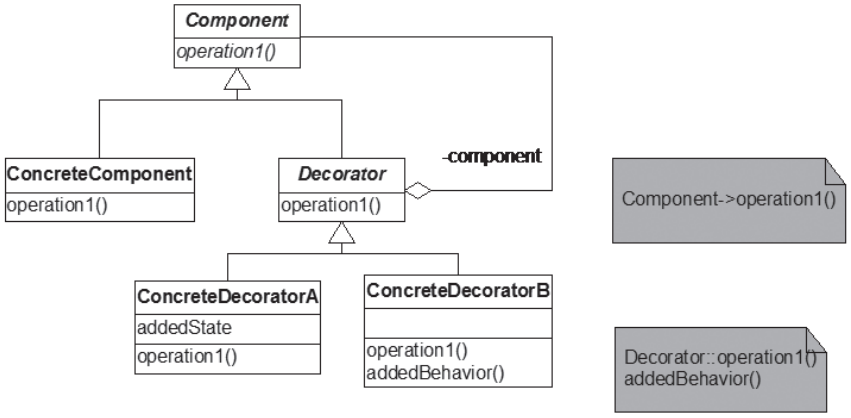

Figure 5. The structure of the decorator pattern.[11]

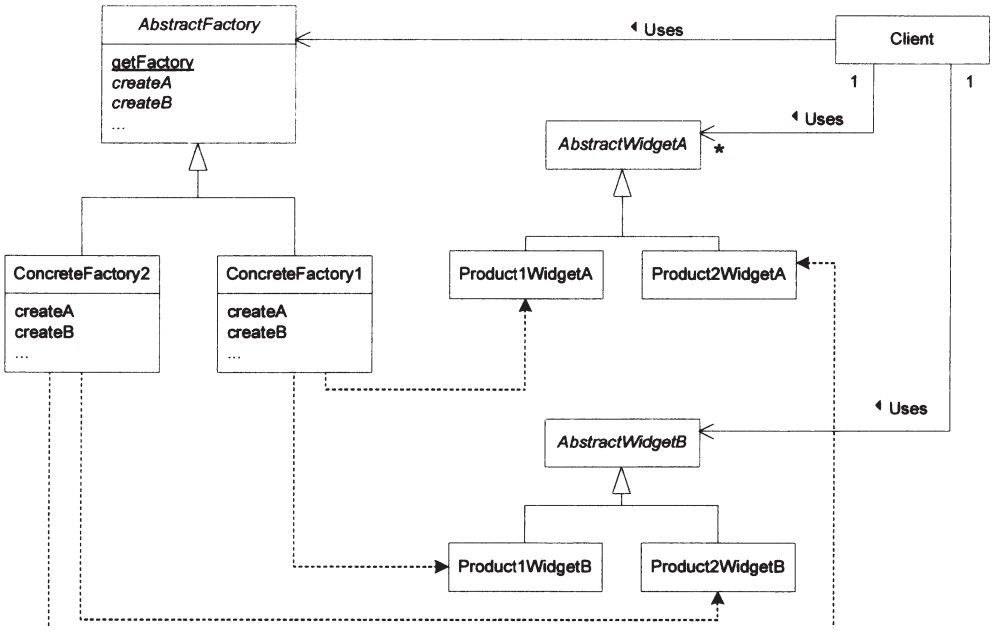

Figure 6. The structure of the abstract factory pattern.[11]

Finally, the Abstract Factory pattern provides an interface for creating families of related or dependent objects without having to specify their concrete (specific) class. The Abstract Factory pattern is useful when there are multiple "families" of products and their implementation details must be hidden. Figure 6 shows the generic structure of the Abstract Factory pattern.

In the next section we will show how these patterns were used to restructure the Kahindu imaging system to eliminate the negative characteristics previously identified.

Refactored Design

To improve the overall design of the Kahindu system we started by considering its basic requirements. The refactored design was then driven by those requirements. The basic requirements of the system are as follows. First, the user selects an image that is displayed by the system. The image may be stored in various image file formats, including JPG, GIF, and PPM. The user may transform the image in a number of ways, such as making the image darker or lighter. Other transformations may include converting a color image to grayscale, or creating a negative image. The user may apply one of several noise reduction filters, or may apply a sharpening filter to bring out details in an image. Finally, the user may apply one of several edge detection algorithms to outline the boundaries of objects in the image.

The transformation of images is a sequential process. The user may make the image darker (one or more times), convert the darkened image to grayscale, convert that grayscale image to a negative, and so on. Also, the user may navigate backwards through the chain of transformations to a particular point in the chain to undo a sequence of changes, or the user may simply want to begin again with the original image. This dynamic layering of views is also common in applications involving text formatting.[12]

These are the basic functional requirements of the image processing system. An important non-functional requirement that will impact the refactored design of the system is the need for flexibility. Future requirements may include the ability to handle additional image file formats, additional filters, and different transformation algorithms. The system needs to be designed so that these extensions can be made with a minimum of

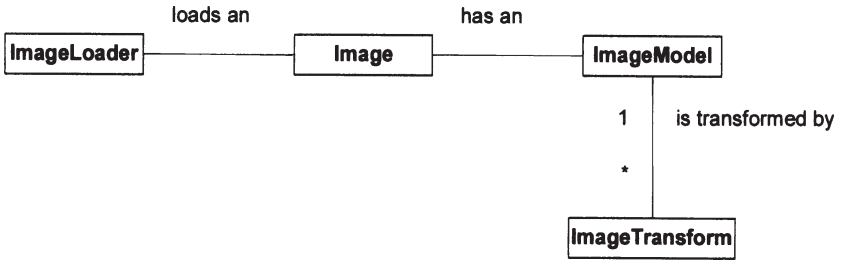

Figure 7. Domain model for image processing system.

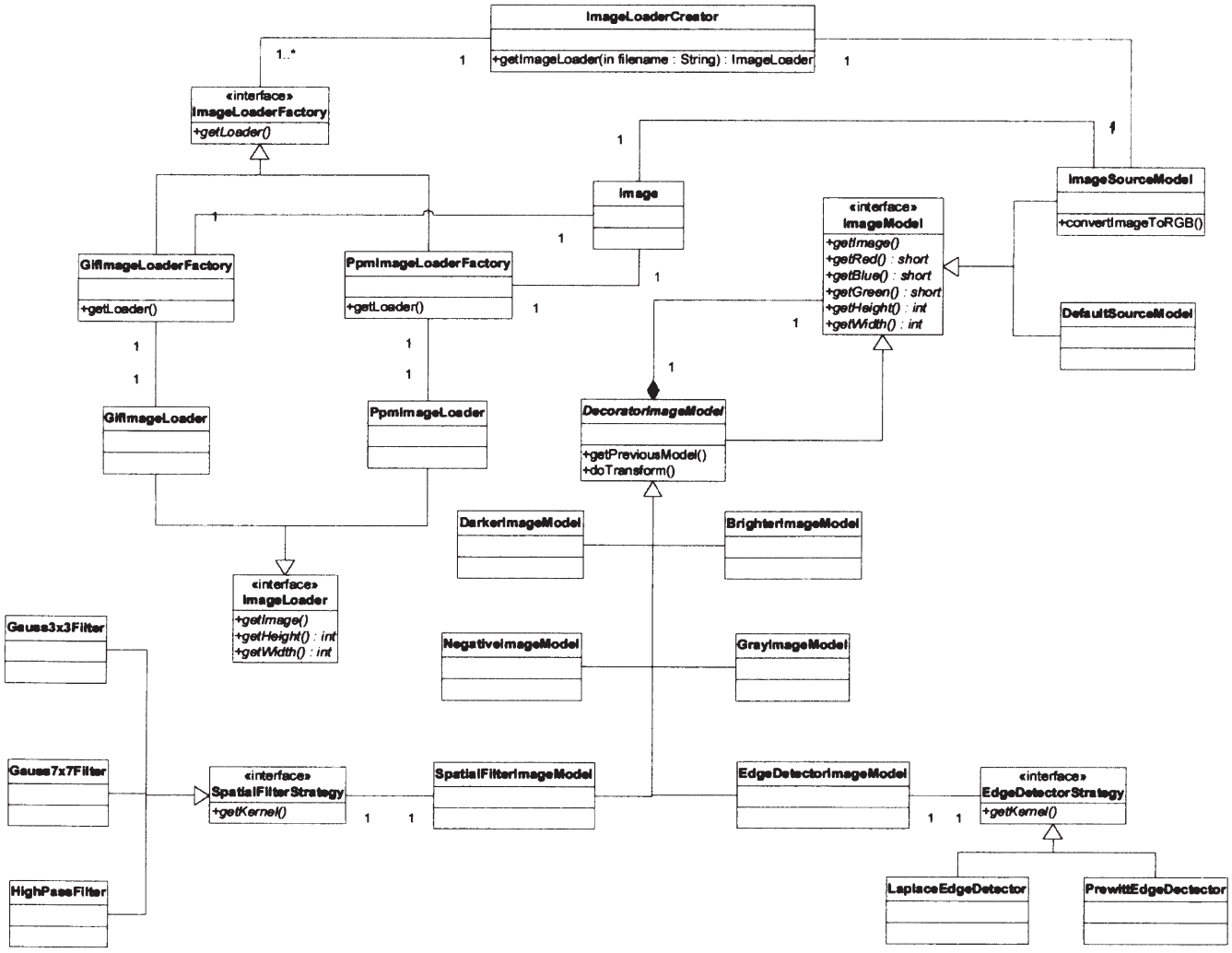

Figure 8. Design class diagram for image processing system.

changes to the existing code. The need to be easily adapted due to frequent changes will fundamentally affect the refactored design.

The domain model of the image processing system is shown in Fig. 7. A domain model is a visual representation of real-world conceptual classes in the problem domain. It reflects the abstractions, vocabulary and information content of the domain. It is not a representation of software objects. The domain model is a fundamental aspect of object-oriented analysis.[13]

The fundamental conceptual classes of the image processing system are an image, an image model, an image loader, and an image transformer. The `Image` class is the internal, packed pixel representation of a graphical image. The `ImageLoader` loads the Image from a storage device. The `ImageModel` is extracted from the `Image`. The `ImageModel` is the representation of the unpacked, red, green and blue color plane arrays that can be transformed by an `ImageTransform` and can be converted to an `Image` for display. The state changes of domain objects that result from system operations are also defined in this stage of the analysis. The domain model narrows the gap between a conceptual model of the domain and a software representation of the domain.

The assignment of responsibilities to classes and objects leads to the class diagram for the image processing system shown in Fig. 8. Responsibilities of

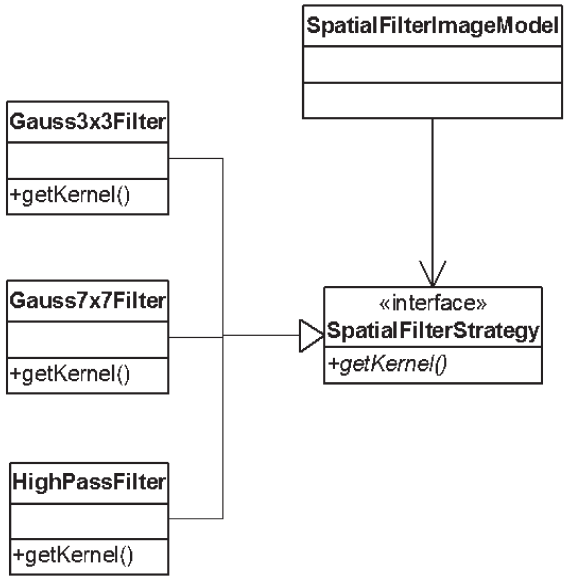

Figure 9. Implementation of strategy pattern for spatial filters.

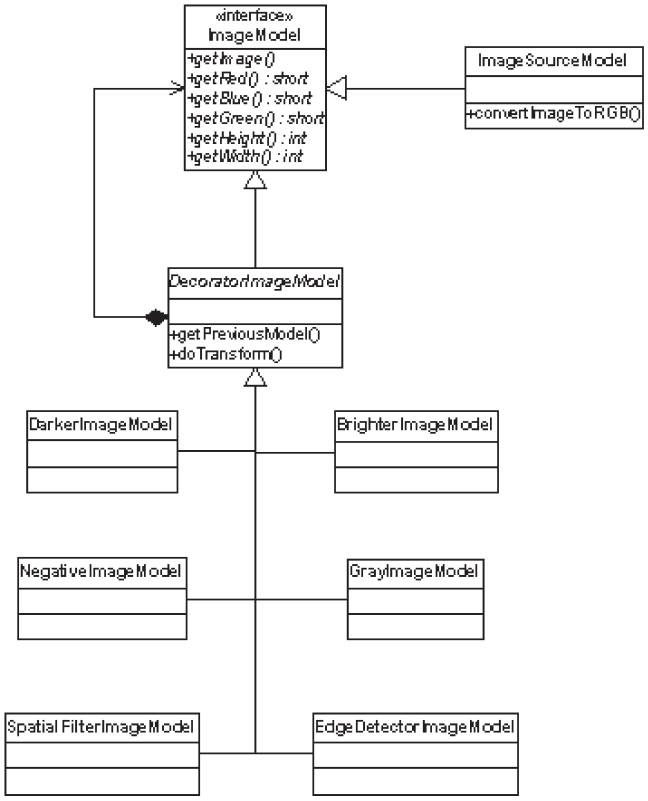

Figure 10. Implementation of decorator pattern in image processing system.

an object may include performing a calculation, creating an object, or collaborating with other objects. The principles for assigning responsibilities to objects are expressed as patterns.

Several patterns have been applied in the refactored design. For example, consider the application of the Strategy pattern to the design of spatial filters shown in Fig. 9. The Strategy pattern separates behaviors from an object, and represents these behaviors in a separate class hierarchy. These behaviors can be flexibly plugged in during runtime. Additional behaviors can be easily added without changing any of the other classes by deriving a new subclass of `SpatialFilterStrategy` class that encapsulates the additional variant behavior.

In the image processor, the primary difference between spatial filters is that each has a unique convolution kernel. By applying the Strategy pattern, the various convolution kernels can be implemented as separate classes, each with a common interface. The appropriate convolution kernel can then be referenced at runtime, and additional convolution kernels can be added without any code changes.

The implementation of the Strategy pattern is as follows. The `SpatialFilterImageModel` is the class that uses the different strategies for different filtering tasks. The `SpatialFilterImageModel` keeps a reference to the Strategy instance that has been specified at runtime. The `SpatialFilterStrategy` is an interface that defines the `getKernel()` method that all `SpatialFilters` must implement. Each `SpatialFilter`, such as the `HighPassFilter`, implements a `getKernel()` method that returns its unique kernel. To implement a different `SpatialFilter`, the new class implements its kernel and none of the other classes need to be changed.

Multiple `EdgeDetector` strategies have also been implemented using a similar Strategy pattern, as can be seen in Fig. 8. The primary benefit of the Strategy pattern, that makes it useful in image processing applications, is the ability to extend the system without extensive recoding.

The refactored image processing system also uses the Decorator pattern to provide the functionality for chaining together various image transformations. This pattern allows layers to be added to and removed from a base object. The Decorator pattern is appropriate for applications which use dynamically built overlays and views, as is the case in the image processing system. The layers can be chained, which provides complex object behavior from a set of fairly simple building blocks.

The implementation of the Decorator pattern is shown in Fig. 10.

The `DecoratorImageModel` is an abstract class that defines the behavior of Decorators. The `DecoratorImageModel` class maintains a reference to the `DecoratorImageModel` that is being transformed. The `DecoratorImageModel` has an abstract `doTransform()` method which is implemented by all `DecoratorImageModels`. The base `ImageModel` is an `ImageSourceModel`, which is the root of the decorator chain. The `ImageSourceModel`, is an `ImageModel` loaded from a storage device. By using the Decorator pattern, coding is simplified, since a series of relatively simple, lightweight decorator classes, each with very specific behavior, is used instead of a complex, heavyweight class that contains all the possible decorator behaviors. Also, like the Strategy pattern, additional behaviors can be added without recoding any of the existing classes. The flexibility of the system is further enhanced by combining the Decorator pattern with the Strategy pattern, as is the case with the `SpatialFilterImageModel` and the `EdgeDetectorImageModel` that were described previously.

Another pattern that is used to provide flexibility and extensibility is the Abstract Factory pattern. The

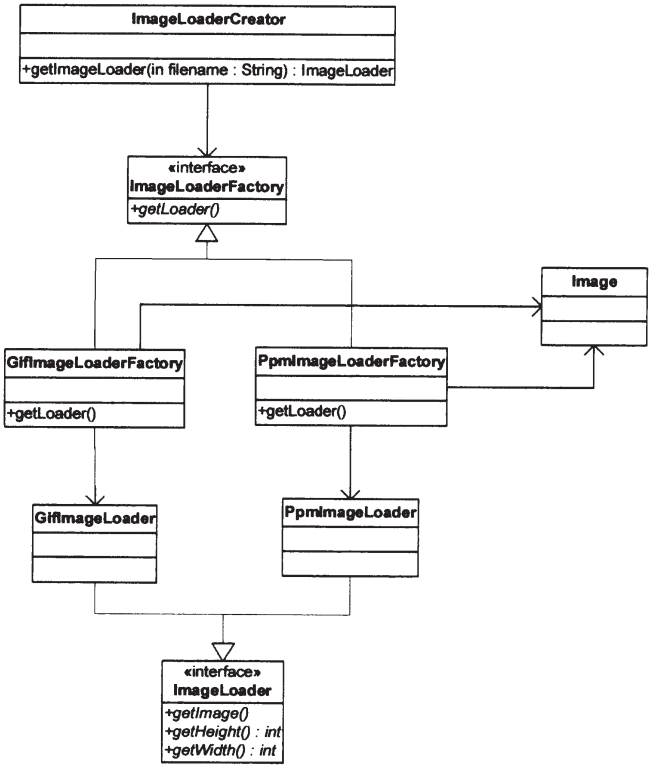

Figure 11. Implementation of abstract factory pattern for image loaders.

image processing system supports various image file formats, and may need to support additional formats at some time in the future. One way to provide this flexibility is to use an interface for the various `ImageLoaders`. However, it is not possible to create an object instance from an interface, so the application needs to instantiate specific implementations of the `ImageLoader` interface. In this case, a code change would be required when a new image file format is added to the system. Alternatively, an object factory can be used to instantiate concrete classes that implement the `ImageLoader` interface.

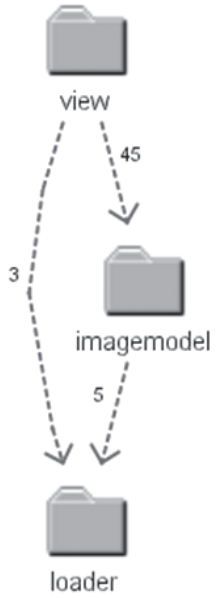

Figure 12. Package dependency high graph for the refactored system.

The implementation of the Abstract Factory pattern in the image processing system is shown in Fig. 11. The `GifImageLoaderFactory` and `PpmImageLoaderFactory` are the classes that create specific implementations of the `ImageLoader` interface, specifically the `GifImageLoader` and the `PpmImageLoader`. The specific type of `ImageLoader` to create is determined by the `ImageLoaderCreator`, which maintains a list of file extensions and the corresponding factory instance for each file extension. Thus, a new image format such as TIF can be added to the system with little change to the existing code.

The design of the refactored image processing system emphasizes flexibility to add functionality, algorithms, and image formats while minimizing code changes and maximizing code reuse. The use of patterns has helped to accomplish these design objectives. The patterns used in this application have also led to the design of

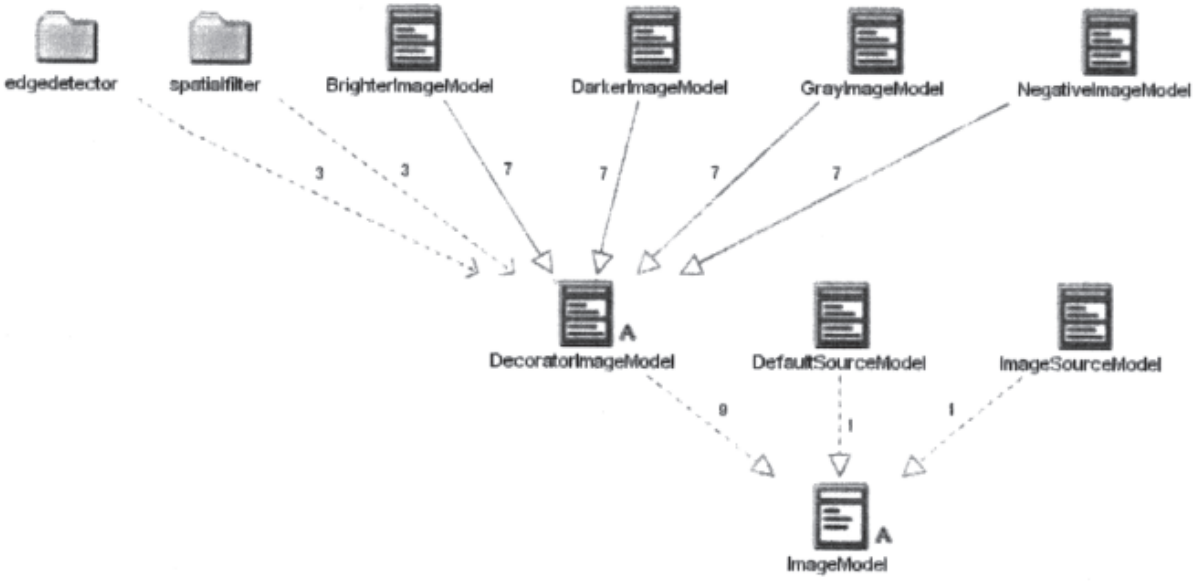

Figure 13. The `imagemodel` package.

TABLE I. Comparative Analysis of Kahindu and the Refactored System

Object-Oriented Metric	Kahindu	New System	Analysis
Cyclic dependencies among packages	Yes	No	Cyclic dependencies make a system more rigid, fragile and difficult to reuse
Response factor for a class	Average: 13.503 Maximum: 103	Average:7.31 Maximum: 54	Classes with high response factor are more difficult to understand, test & debug.
Depth of inheritance hierarchy	Average: 1.567 Maximum:19	Average: 0.207 Maximum: 1	Large depths of inheritance hierarchy imply complex design that is harder to understand and test
Data classes	25	1	Data classes violate object-oriented design
Classes with feature envy methods	8	0	Feature envy methods break encapsulation
Large classes (measured through method count or MC)	Avg. MC: 9.737 Maximum MC: 75	Avg. MC: 5.44 Maximum MC: 35	Large classes have too many responsibilities

Figure 14. The images used for performance analysis.

lightweight, loosely coupled, highly cohesive classes, which improves the maintainability, reliability and integrity of the system.

Analysis and Comparison of the Restructured Design

We performed an analysis similar to the one for Kahindu on the refactored design. Figure 12 shows the first snapshot the new system.

The `loader` package is assigned the responsibility of loading images from various file formats. The `imagemodel` package transforms the loaded image into an internal representation for ease of manipulation. The `view` package displays the image. The packages have no cyclic dependencies. Figure 13 shows a snapshot of `imagemodel`, the most complex package in this system.

The `edgedetector` and `spatialfilter` subpack-ages contain four or five classes each for edge detection and spatial filtering strategies. The `imagemodel` pack-age is, therefore, much simpler when compared to the `gui` package of Kahindu. Table I provides a summary of the comparison between Kahindu and the refactored system.

Performance Analysis

The performance analysis was performed on a 1.79 GHz Intel Pentium 4 with 1.0 GB of RAM running under Microsoft Windows XP. We ran ten iterations of Kahindu and the refactored system on each of the three commonly used images, "Baboon", "Lena", and "Peppers" shown in Fig. 14. The images from each of the ten trials for each system and transformation were identical since there is no variation in the algorithm from trial to trial. The only variation in performance for the ten trials is due to the varying time needed for loading classes, garbage collection, and I/O.

The result of the analysis is shown in Fig. 15. The title of each graph indicates the operation, the Y-axis gives the execution time for each operation in milliseconds and the X-axis shows the iterations. Iterations 0 to 10 are timing results for Baboon, 11 to 20 for Lena and 21 to 30 for Peppers. As can be seen, the refactored system consistently outperformed Kahindu for all operations on all images.

We had anticipated that while using design patterns would make our solution more understandable and maintainable, it would introduce layers of indirection slowing down the system. However, the results turned out to the contrary. Our experiments showed real-time performance improvements in the range 8% to 87% for the refactored system over the original system, with a mean improvement of 44% and a median of 47%.

The improved performance of the refactored solution can be attributed to a number of factors. Primarily, however, we believe the use of design patterns simplified the solution and in the process may have eliminated needless indirection. For example, the original Kahindu system incorporated very deep inheritance hierarchies and therefore significant indirection. These inheritance hierarchies are also suggestive of heavy usage of polymorphic code. Polymorphism, while allowing one to write highly dynamic code, can be expensive with respect to performance. We intend to do further analysis of the Kahindu code in our future work to get a better understanding of these results.

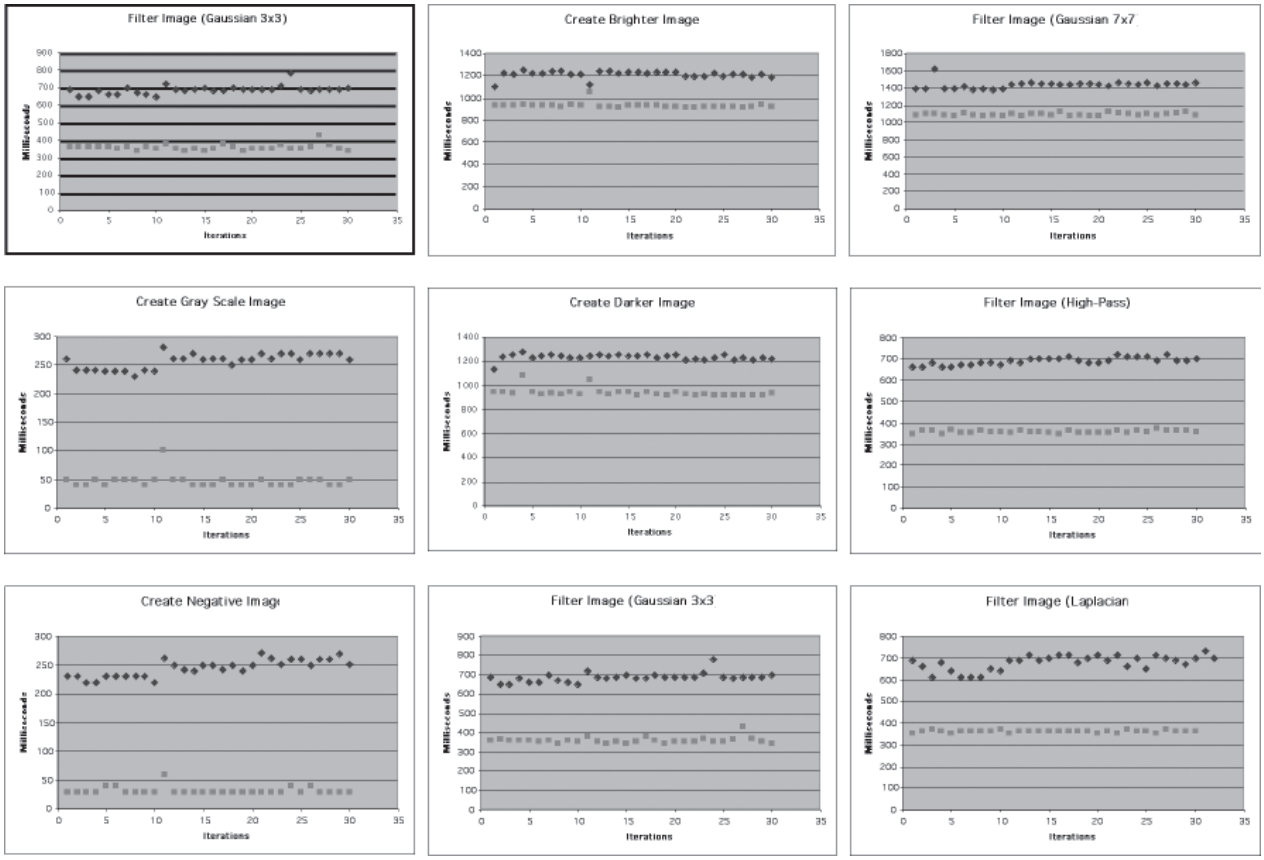

Figure 15. Performance analysis results of Kahindu (upper) and the refactored system (lower).

Conclusions

Object-oriented design has long been believed to provide substantial benefits in terms of code maintainability, understandability, reusability, and many other qualities. But imaging engineers have tended to avoid object-oriented designs because of the belief that such code had to have poor real-time performance. This work, however, shows that the benefits of reusability and good real-time performance can be had when best design practices, in particular, the use of patterns are followed. The work also suggests general guidelines to follow when restructuring an existing system into a well designed object oriented system. The domain model derived from the requirements of the system under consideration is used as a motivator for the software classes. The software classes are then systematically assigned responsibilities using the software design patterns.

We have made an observation that depth of inheritance, and interdependencies among software packages and classes influence performance. We intend to study the feasibility of quantifying the impact of these qualities from structural analysis of the system under consideration.

Future work also includes comparing performance between well written object-oriented code and structured code using a procedural paradigm. Structured programming using abstract data types has been shown to result in systems with considerably smaller code size and overhead. We will examine the extent of reuse provided by such systems when compared to well-written object-oriented code.

References

1. P. A. Laplante and C. J. Neill, Software specification and design for imaging systems, *J. Electronic Imaging* **12**(2), 252-262 (2003)
2. C. J. Neill, Leveraging object-orientation for real-time imaging, *Real-Time Imaging* **9**(6), 425-434 (2003).
3. C. J. Neill and P. A. Laplante, Image frameworks: Design for reuse in real-time imaging, *Proc. SPIE*, SPIE, Bellingham, WA, 2004.
4. D. Lyon, *Image Processing in Java*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.
5. Headway reView, http://www.headwaysoft.com.
6. T. McCabe and C. Butler, Design complexity measurement and testing, *Communications of the ACM* **32**(12), 1415-1425 (1989).
7. S. Chidamber and C. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Software Eng.* **20**(6), 476-493 (1994).
8. R. C. Martin, OO Design Quality Metrics: An Analysis of Dependencies, ROAD, 1995, http://www.objectmentor.com/resources/articles/oodmetrc.pdf.
9. J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, New York, NY, 1996.
10. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, New York, NY, 1999.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, NY, NY, 1995.
12. S. Stelting and O. Maassen, *Applied Java Patterns*, Sun Microsystems Press, Palo Alto, CA, 2002.
13. C. Larman, *Applying UML and Patterns*, Prentice-Hall, Upper Saddle River, NJ, 2002.