Software Requirements Specification for Imaging Systems

Phillip A. Laplante and Colin J. Neill

Vol. 12, No. 3
April 2003

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
Software specification and design for imaging systems

Phillip A. Laplante
Colin J. Neill
The Pennsylvania State University
Malvern, Pennsylvania 19355

Abstract. Although it would seem obvious that object-oriented techniques are well suited to modeling image processing systems, there is almost no research to substantiate this position. We review the unique challenges faced when modeling, that is, specifying and designing, imaging systems. First, the argument is presented that specification and design of imaging systems should be studied as a special case. Next, a subset of techniques that have been used to model practical imaging systems is surveyed. Then the use of object-oriented techniques in the specification of imaging systems is investigated more fully and compared to a widely used alternative—structured analysis and design—by way of a case study. We conclude with some recommendations for best practices and future research. © 2003 SPIE and IS&T. [DOI: 10.1117/1.1557154]

1 Why Image Processing Specification and Design Is Different

Software modeling techniques can be classified as operational, descriptive, or dual if they have both operational and descriptive capabilities. Operational techniques are those that are defined in terms of states and transitions. Descriptive techniques are based on mathematical notation. Dual techniques tend to integrate the characteristics of both, allowing the formal specification to be both high and low level.

Imaging engineers tend to widely mix operational and descriptive specifications and to vary the level of detail within these (mixing instructions, procedures, and systems), so as to obfuscate any notion of leveling (providing a uniform level of abstraction) within the system specification. This is not generally the case in other kinds of systems, even other types of embedded systems.

But there are other compelling reasons as to why imaging systems are different from other kinds of systems. These reasons are mainly that they involve the following volatile components:

1. interchangeable algorithms for critical operations (e.g., compression/decompression, filtration, enhancement, and display),
2. rapidly changing underlying hardware (e.g., displays, cameras, and storage),
3. a great deal of hardware and software reuse, and
4. legacy and off-the-shelf software that has not been developed using rigorous software engineering approaches.

While these characteristics can sometimes be found in other kinds of systems, they are pervasive in imaging systems.

Unfortunately, there is little literature available that treats the specification or design of imaging systems as a special case. A thorough discussion of specification techniques for generalized complex applications is given in Ref. 3, but this study focuses largely on the connection between the specification technique and the programming language. Moreover, the unique aspects of modeling imaging systems, for example, how to represent images, is not covered. A wide-ranging, historical review of specification techniques is given in Ref. 1, but it is more focused on generalized specification methods and not on languages. More recent work by the authors has focused on these neglected topics, including a survey of techniques used in modeling image processing systems, a study of the use of object-oriented methods for modeling imaging systems both with and without temporal constraints, and the use of Unified Modeling Language (UML) to specify real-time imaging systems. Many of these results are included and expanded on in this work.

2 Survey of Commonly Used Techniques

There appears to be no comprehensive study of specification approaches for imaging applications. Existing surveys of requirements’ specifications do not address the use of these approaches in imaging applications. In general, it seems that imaging engineers tend to use one or a combination of the following approaches in writing software specifications for imaging applications. These are:

• top-down process decomposition or structured analysis (structured analysis and structured design for example)
• object-based or object-oriented approaches
• program description languages (PDL) or pseudo-code
• high-level functional specifications that are not further decomposed
• ad hoc techniques, including simply natural language
and mathematical description, that are always included in virtually every system specification.

To illustrate this, several fragments of examples are given in the following sections.

2.1 Multiresolution Block Matching System Specification Using a Block Diagram and Flowchart

The example shown in Fig. 1 from Shi and Sun\(^7\) indicates the use of flow charts in the operational specification for a multiresolution block-matching algorithm such as those used in motion estimation. In such a system, an image is partitioned into a set of nonoverlapped, equally spaced, fixed size, and small rectangular blocks. Then, displacement vectors for these blocks are estimated by finding their best-matched counterparts in the previous frame. In this manner, motion estimation is significantly easier than that for arbitrarily shaped blocks.

A procedural description of the block-matching component denoted in Fig. 1 is then depicted using a flow chart, as shown in Fig. 2. No further specification is given for the technique, and it is expected that an “ordinary” imaging engineer would be able to implement a design and build the systems using this level of specificity. There is no consideration for the overall software architecture, the data structures and their relationships, or the timing constraints that must be met. While this technique is suitable for the specification of individual tasks or algorithms, it is not scalable for large systems.

2.2 Collision Testing of Graphical Objects Using Pseudo-Code

Specification techniques with similar problems of scalability are pseudo-code and program description languages. An example of a pseudo-code specification is shown in Fig. 3. This is for the collision testing between graphical objects frequently used in video games taken from Möller and Haines.\(^8\) This type of specification is again sufficient for individual tasks or algorithms, but only at late stages of design where the majority of decisions have been made, because the pseudo-code representation is so close to code that it will likely be regarded as the final design version of system behavior, rather than a description of system expectation.

2.3 Functional Representation of Machine Vision System Using a Structured Approach

Rajeswari and Rodd\(^9\) use a functional description, similar to a context diagram used in structured design, to describe a machine vision system for inspecting flaws in integrated circuit wire bonds. The functional description is given in Fig. 4. Further decomposition is given using their Q-model, which is a novel technique for the specification of temporal properties of systems similar to Petri nets.

This represents a significant advance over the previous examples, since there is a segregation of perspectives in the overall specification: data flow is indicated in one diagram and timing and control information is on another. The use of

\[\text{FindFirstHitCD}(A, B)\]

\[
\text{returns} (\{\text{TRUE}, \text{FALSE}\});
\]

1: \(\text{if (not overlap}(A_{BV}, B_{BV}) \text{ return FALSE;})\)
2: \(\text{else if (isLeaf}(A)\))
3: \(\text{if (isLeaf}(B)\))
4: \(\text{for each triangle pair } T_A \in A_c \text{ and } T_B \in B_c\)
5: \(\text{if (overlap}(T_A, T_B)\)) \text{ return TRUE;}
6: \(\text{else}\)
7: \(\text{for each child } C_B \in B_c\)
8: \(\text{FindFirstHitCD}(A, C_B)\)
9: \(\text{else}\)
10: \(\text{for each child } C_A \in A_c\)
11: \(\text{FindFirstHitCD}(C_A, B)\)
12: \(\text{return FALSE;}
\]

Fig. 3 Hierarchical collision testing algorithm.
of structured approaches has its own drawbacks, however, which is discussed later with reference to a case study.

2.4 Markov Random Fields Image Reconstruction Using Object-Oriented Design

The final example found in the literature is the use of object-oriented techniques. It is important to note that there is a distinction between object-oriented design and specification and object-based image processing techniques. The latter is a technique used in image processing for identifying visual objects in a scene based on certain features. This is clearly different from the former, which is a software engineering approach interested in modeling systems as communities of entities that encapsulate state and behavior. However, it seems that object-oriented techniques do seem to be well suited to specifying object-based imaging approaches.

For example, Mariatos et al.\textsuperscript{10} proposed an object-oriented design for an image reconstruction system used to reconstruct images damaged by transmission errors or lossy compression. The system applies edge-preserving smoothing transforms on an input image. The outputs are then a surface field (the transformed image) and a discontinuity field (the edge information). It can be seen from the class model in Fig. 5 that rather than decompose the system into constituent processes—a characteristic present in all the previous examples—the model is formed by representing the entities involved (F Field is the surface field, for example). This aspect of object-oriented analysis and design is studied in detail further along in the work.

2.5 Formal Methods

Formal methods attempt to improve requirements' formulation and expression by the use and extension of existing mathematical approaches such as propositional logic, predicate calculus, and set theory. This approach is attractive because it offers a highly scientific way to requirement specification. Writing formal requirements can often lead to error discovery in the earliest phases of the software lifecycle, where they can be corrected quickly and at a low cost. Informal specifications might not achieve this goal because they are not precise enough to be refuted by counterexample.\textsuperscript{1}

By their nature, specifications for most imaging systems usually contain some formality in the mathematical expression of the underlying imaging operations. While this fact does not justify claiming that any imaging system specification is fully formalized, it does lead to some optimism that imaging systems can be made suitable for at least partial formalization.

Formal methods, however, are difficult to use by even the most expertly trained and are sometimes error-prone. For these reasons and because they are sometimes perceived to increase early lifecycle costs and delay projects, they are frequently avoided.

Approaches to requirement specification that are not formal, are either informal (such as the techniques discussed in Secs. 2.1 through 2.4) or semiformal. The UML is a semiformal specification approach, meaning that while it does not appear to be mathematically based, it is, in that every one of its modeling tools can be converted to an
underlying mathematical representation. Hence, it enjoys the benefits of both informal and formal techniques.

3 Case Study

The previous survey of requirement specification techniques used indicates that there is certainly no standard way to write requirements for imaging systems. Moreover, most of the techniques that are used are informal or even ad hoc. It can be concluded, then, that much of what is going on in the engineering of imaging systems lacks software engineering rigor. Because of this lack of rigor, many systems are probably costing more than they should to build and maintain.

The lack of a de facto standard technique for specification and design further raises the question of whether any technique is more appropriate than another for the modeling of an imaging system. This work takes the position that object-oriented techniques are appropriate for specification and design of imaging systems, and in fact, the semiformal UML is a most desirable way to develop object-oriented software requirements specifications in this regard. To further explore this hypothesis, an example is presented of the specification of an imaging system using both an object-oriented technique and non-object-oriented counterpoint, structured analysis and structured design (SASD).

The example is adapted from Ref. 11 and is of an industrial automated visual inspection (AVI) system. AVI is an interesting case, since it represents a simple intuitive example of an embedded system where the temporal performance is dictated by the operating environment rather than by the computer system itself. A typical setup is shown in Fig. 6. Images of the products are captured as they move along the conveyor. The images are preprocessed and then classified, using an appropriate feature-matching algorithm, as pass or fail by the system, and defective products are removed from the conveyor by the reject mechanism. The system is further detailed as the discussion proceeds.

3.1 Structured Analysis and Design

Methods for SASD have evolved for nearly 30 years and are widely used in image processing applications, probably because the techniques are closely associated with the programming languages with which they coevolved (Fortran and C), and in which many image processing applications are written. Structured methods appear in many forms (see Refs. 12–15) but the de facto standard is Yourdon’s modern structured analysis.

Yourdon’s modern structured analysis uses three viewpoints to describe a system: an environmental model, a behavioral model, and an implementation model. The elements of each model are shown in Fig. 7.

The environmental model embodies the analysis aspect of SASD and consists of a context diagram and an event list. The purpose of the environmental model is to model the system at a high level of abstraction.

The behavioral model embodies the design aspect of SASD as a series of data flow diagrams (DFDs), entity relationship diagrams (ERDs), process specifications, state transition diagrams, and a data dictionary. Using various combinations of these tools, the designer models the processes, functions, and flows of the system in detail.

Finally, in the implementation model, the developer uses a selection of structure charts, natural language, and pseudo-code to describe the system to a level that can be readily translated to code.

3.1.1 Structured analysis

Structured analysis (SA) is a way to try to overcome the problems of classical analysis using graphical tools and a top-down, functional decomposition method to define system requirements. SA deals only with aspects of analysis that can be structured—the functional specifications and the user interface.

SA is used to model a system’s context (where inputs come from and where outputs go), processes (what functions the system performs, how the functions interact, and how inputs are transformed to outputs), and content (the data the system needs to perform its functions).

SA seeks to overcome the problems inherent in analysis through:

- maintainability of the target document,
- use of an effective method of partitioning,
- use of graphics,
- building a logical model of the system for the user before implementation, and
- reduction of ambiguity and redundancy.

The target document for SA is called the structured speci-
fication. It consists of a system context diagram, an integrated set of data flow diagrams showing the decomposition and interconnectivity of components, and an event list to represent the set of events that drive the system.

To illustrate the SA technique, consider the visual inspection system previously introduced. Figure 8 depicts the context diagram. Here, the visual inspection system is shown with the other constituent system parts—camera, product detector, production conveyor controller system, and reject mechanism. Solid arcs indicate the flow of data between system components. In the example the only data flow involves the transmission of the captured image to the visual inspection system. The dashed lines represent the flow of control information. This facility is one of the extensions needed for dealing with real-time systems, which is discussed in Ref. 5.

In the example, the event list consists of the new_product_event, which indicates the detection of the next image on the line; accept, which indicates that the product has passed inspection and causes a signal to be sent to the conveyor controller; and reject, which causes a signal to be sent that directs the conveyor to move the product into a rejected product bin. The rejection mechanism automatically causes the next product item to be moved along by the conveyor controller.

### 3.1.2 Structured design

Structured design (SD) is a systematic approach concerned with the specification and design of the software architecture and involves a number of techniques, strategies, and tools. SD provides a step-by-step design process that is intended to improve software quality and reduce risk of failure, and increase reliability, flexibility, maintainability, and effectiveness.

SA is related to SD in the same way that a requirements representation is related to the software architecture, that is, the former is functional and flat and the latter is modular and hierarchical. This relationship is similar to the comparison between a state transition diagram and a structure chart (see Fig. 9).

The transition mechanisms from SA to SD are manual and involve significant analysis and trade-offs of alternative approaches. Normally, SD proceeds from SA in the following manner. Once the context diagram is drawn, a set of data flow diagrams is developed. The first data flow diagram, the level 0 diagram, shows the highest level of system abstraction. Decomposing processes to lower and lower levels until they are ready for detailed design renders new DFDs with successive levels of increasing detail. This decomposition process is called leveling.

In a typical DFD, boxes represent terminators that are labeled with a noun phrase that describes the system, agent, or device from which data enters or to which data exits. Each process, depicted by a circle, is labeled as a verb phrase describing the operation to be performed on the data, although it may be labeled with the name of a system or operation that manipulates the data. Solid arcs are used to connect terminators to processes and between processes to indicate the flow of data through the system. Each arc is labeled with a noun phrase that describes the data. Dashed arcs are discussed later. Parallel lines indicate data stores, which are labeled by a noun phrase naming the file, database, or repository where the system stores data.

Each DFD should have between three and nine processes only. The descriptions for the lowest level, or primitive, processes are called process specifications, or P-SPECs, and are expressed in either structured English, pseudo-code, decision tables, or decision trees and are used to describe the logic and policy of the program.

Returning to the visual inspection system example, Fig. 10 shows the level 0 DFD.

Here the details of the system are given at a high level. First, the system reacts to the arrival of a new product by...
confirming that the image data is available. Next, the system captures the image by buffering the raw data from the capture device to a file. Preprocessing of the raw data is performed to produce an image frame to be used for classification and generation of the appropriate control signals to the conveyor system.

Proceeding to the next level provides more detail for processes 1, 2, 3, and 4. Process 1 is essentially an interrupt service routine assigned to a photodiode detector that senses when a new product for inspection reaches the designated point on the conveyor. Process 2 is a buffering routine, whose characteristics depend on the specifications of the camera. Hence, without knowing these details, it is not possible to go deeper into the design.

Figure 11 depicts the level 1 DFD for process 3. Notice how the internal processes 3.1 and 3.2 are labeled to denote that they are a finer degree of detail of process 3 shown in the 0 level diagram. Successive levels of detail will follow a similar numbering system, e.g., 3.1.1, 3.1.2. This convention provides simple traceability from specification through design and on to code. Proceeding with the design example, Fig. 12 shows the level 1 DFD for process 4 without further explanation.

In addition to the DFDs, SD uses a data dictionary to document and control interfaces. Entity relationship diagrams are frequently used to define the relationship between the components of the system, much as in the object-oriented paradigm. The data dictionary documents each interface flow in the DFD. Data structure diagrams are also used to describe information about logical relationships in complex data structures. The entity relationship model (which is optional) and data dictionary for the visual inspection system are not shown for brevity.

### 3.1.3 Problems with SASD in imaging applications

There are several apparent problems in using SASD to model the visual inspection system, including difficulty in modeling time and events. For example, what if the visual inspection system captures a new image in parallel with preprocessing of the last image capture? (This scenario would be desirable if the reject mechanism were further down the inspection line and the conveyor system were running at a high rate.) Concurrency is not easily depicted in this form of SASD.

Another problem arises in the context diagram. Control flows are not easily translated directly into code, such as “reject” and “accept,” because they are hardware dependent. In addition, the CFD does not really make sense, since there is no connectivity between portions of it, a condition known as “floating.”

Details of the detector and camera hardware also need to be known for further modeling of process 1. What happens if the hardware changes? What if a different strategy for classification in process 2 is needed? In the case of process 3, (preprocessing), what if the algorithm or even the sensitivity levels change because of the installation of new hardware? In each case the changes would need to propagate into the level 1 DFD for each process, any subsequent levels, and, ultimately, into the code.

Clearly making and tracking any of these changes is fraught with danger. Moreover, any change means that significant amounts of code would need to be rewritten, recompiled, and properly linked with the unchanged code to make the system work. None of these problems arise using the object-oriented paradigm.

### 3.2 Object-Oriented Analysis and Design

There are various “flavors” of object-oriented analysis and design (OOAD), each using their own toolsets. The development process described in Ref. 17 was used for the case study, and several of the UML models generated are shown. In this approach the system specification begins with the representation of externally accessible functionality as use cases.

Use Cases are an essential artifact in OOAD and are described graphically in one of the nine models provide by the UML. The use case diagram can be considered analogous to the context diagram in SASD, in that it represents the interactions of the software system with its external environment.

Use cases are represented graphically as ellipses, as can be seen in Fig. 13. Each use case is, however, a document that describes scenarios of operation of the system under consideration as well as pre- and post-conditions and exceptions. In an iterative development lifecycle, these use cases become increasingly refined and detailed as the analysis and design workflows progress. Interaction dia-
grams are then created to describe the behaviors defined by each use case. In the first iteration, these diagrams depict the system as a "black box," but once domain modeling has been completed, the black box is transformed into a collaboration of objects as is seen later.

As stated before, the domain model is created based on the use cases and, through further exploration of system behavior via the interaction diagrams, the domain model evolves systematically into the design class diagram. The construction of the domain model is, therefore, analogous to the analysis stage in SASD described earlier. In domain modeling the central objective is to represent the real-world entities involved in the domain as concepts in the domain model. This is a key aspect of object-oriented systems and is seen as a significant advantage of the paradigm, since the resultant model is closer to reality than in alternative modeling approaches, including SASD. Part of the design class diagram that results from evolution of the domain model is shown in Fig. 14.

The design class diagram is used to show the static structural view of the system by describing the classes of objects that comprise the software solution. As can be seen in Fig. 14 the system is composed of image capture elements (CameraProxy, FrameGrabber) and image classification elements (Classifier, ImageProcessor, decorators, and strategies). The decorators and strategies are aspects of the design introduced by applying well-known design patterns from Ref. 18. The principle aim of these patterns is to allow for dynamic (i.e., runtime) changes to the preprocessing and classification schemes. This is achieved by abstracting invariant behavior into supertypes (ImageProcessor, ProcessDecorator, and ClassifierStrategy) and allowing subtypes to implement variant behavior (N-FeatureComparison, FilterDecorator).

It can be argued that this generality is speculative, and therefore overcomplicates the design, but the intent of this work is to build a framework for image processing systems where this generality is critical for reuse. This also highlights a key advantage with object-oriented analysis and design: the reuse potential. In SASD development it can be very difficult to extend the functionality of the complete system because of the degree and direction of dependencies that are created between high- and low-level modules. That is, high-level modules call lower-level modules as a consequence of the top-down decomposition approach. When changes need to be made at these lower-levels, the more abstract elements must also be modified due to these dependencies. In object-oriented systems this dependency hierarchy is normally inverted.

For example, the N-FeatureComparison classification strategy could involve syntactic pattern recognition, correlation measures, or morphological approaches, which all depend on the internal representation of the image. By subclassing N-FeatureComparison, this extension can be accommodated without modifying any existing code or affecting any other requirements. In the case of the top-down, structured design fostered by SASD, this is not always the case.

![Fig. 13 Use case diagram of VIS.](image1)

![Fig. 14 Partial design class diagram of the VIS.](image2)
case, and in fact, it is likely that every module in the system that used the feature comparison module would need to be rewritten.

Behavioral aspects of the design can be represented by a number of different diagrams in the UML. Perhaps the most popular choice is to use sequence diagrams. The sequence diagram shown in Fig. 15 represents the ordering of messages between objects in the system in response to the arrival of the next product on the conveyor. It is clear that the nextProduct message is generated from the external sensor, which triggers the FrameGrabber. The Image created by the output from the Camera (via the CameraProxy) is preprocessed using one of the decorators (FilterDecorator) and then classified using a classification strategy (N-FeatureComparison). The result of the classification is then sent to the RejectController, which logs the result and triggers the rejection mechanism if required.

4 Object-Oriented Versus Structured Design and an Analysis of the Case Study

The previous observations beg the question: Is OOAD more suitable than SASD for the visual inspection system in particular, and image processing applications in general? SASD and OOAD are often compared and contrasted, and indeed, they are similar in some ways. This should be no surprise, since both have their roots in the work of Parnas and his predecessors.\(^\text{12,13}\) Table 1 provides a side-by-side comparison of the methodologies.\(^\text{19}\)

Both SASD and OOD are full lifecycle methodologies and use some similar tools and techniques. However, there are major differences. SASD describes the system from a functional perspective and separates data flows from the functions that transform them, while OOAD describes the system from the perspective of encapsulated entities that possess both function and form.

Additionally, OOAD models include inheritance while SASD does not, while SASD has a definite hierarchical structure this is a hierarchy of composition rather than heredity. This shortcoming leads to difficulties in maintaining and extending both the specification and design, such as in the case of changes in the visual inspection system example.

The purpose of this discussion is not to dismiss SASD, or even to conclude that it is better than OOAD in all cases. An overriding indicator of suitability of OOAD versus SASD to image processing is the nature of the application. To see this, consider the vertices of the triangle in Fig. 16 representing three distinct viewpoints of a system: data, actions, and events.

Events represent stimuli and responses such as measurements in process control systems, as in the case study. Actions are rules that are followed in complex algorithms, such as “binarize,” “threshold,” and “classify.” The majority of early computer systems were focused on one, or at most two, of these vertices. For example, early, non-real-time image processing systems were data and action intensive but did not encounter much in the way of stimuli and response.

Image processing is data intensive and would seem well suited to SASD. But often the image itself contains control

<table>
<thead>
<tr>
<th>Table 1: A comparison of SASD and OOAD.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SASD</strong></td>
</tr>
<tr>
<td>System components</td>
</tr>
<tr>
<td>Data processes</td>
</tr>
<tr>
<td>Control processes</td>
</tr>
<tr>
<td>Data stores</td>
</tr>
<tr>
<td>Characteristics</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Fig. 15 Sequence diagram of use case product classification.
information (e.g., “reject,” “accept”), which is not well suited to SASD. Moreover, while it is true that image processing is data intensive and an image has high information content, this is not the same as the data intensity found in, say, a database management system. It is likely that image processing is as much event or activity based as it is data based, which makes it quite suitable for object-oriented techniques.

The evidence and case study point out, then, that OOAD should be seriously considered for broader use with image processing systems. Yet, only one (short) work discussing the use of object-oriented approaches in image processing could be found, and it was in the context of teaching image processing with object-oriented programming languages. Further study, practice, and experience are clearly needed.

5 Benefits of Object Orientation

The previous section highlighted some considerations concerning the appropriateness of the object-oriented paradigm to various application areas. There are, however, some additional benefits to using OOAD. When considering the benefits of object-oriented approaches to image processing applications, it is easy to get wrapped up in the ideas of combining data and behavior into an encapsulated entity that better approximates the “things” in our problem domain and consider this closeness between reality and the modeling domain to be the central benefit of using objects. While this can be considered an advantage, the purported intuitiveness of the approach is, in fact, something of a fortuitous side effect. The real advantages of applying object-oriented paradigms are the future extensibility and reuse that can be attained, and the relative ease of future changes.

Several studies indicate that software systems are subject to near-continuous change: requirements change, merge, emerge, and mutate; target languages, platforms, and architectures change; and most significantly the way the software is employed in practice changes. This flexibility places considerable burden on the software design: how can systems that must support such widespread change be built without compromising quality?

While object-oriented systems can be designed to be as rigid and resistant to extension and modification as in any other paradigm, object orientation has the ability to include distinct design elements that can cater to future changes and extensions. These design patterns were first introduced to the mainstream of software engineering practice by Gamma et al. The “Gang of Four” (GoF) patterns, as they are commonly known, are based on four key principles that have been recognized as supporting reuse.

5.1 Open Closed Principle

First recorded by Meyer, the open closed principle (OCP) states that classes should be open to extension, but closed to modification. That is, it should be possible to extend the behavior of a class in response to new or changing requirements, but modification to the source code is not allowed. While these expectations may seem at odds, the key is abstraction. In object orientation, a superclass can be created that is fixed, but can represent unbounded variation by subclassing. This is evident in the prior case study in the classification strategies, where subclasses for each of the various classification algorithms are created, which inherit their interface from an abstract superclass. This aspect is clearly superior to structured approaches and top-down design in, for example, changes in classification strategies, which would require new function parameter lists and wholesale recompilation of any modules calling that code in the structured design.

5.2 Once and Only Once

While certainly not a new idea, Beck put a name to the principle that any aspect of a software system—be it an algorithm, a set of constants, documentation, or logic—should exist in only one place. This isolates future changes, makes the system easier to comprehend and maintain, and through the low coupling and high cohesion that the principle instills, the reuse potential of these aspects increases. The encapsulation of state and behavior in objects, and the ability to inherit properties between classes, allows for the rigorous application of these ideas in an object-oriented system, but is difficult to implement in structured techniques.

5.3 Dependency Inversion Principle

The dependency inversion principle (DIP) states that high-level modules should not depend on low-level modules. Both should depend on abstractions, which can be restated as: abstractions should not depend on details, details should depend on abstractions. Martin introduced this idea as an extension to OCP with reference to the proliferation of dependencies that exist between high- and low-level modules. For example, in a structured decomposition approach, the high-level procedures reference the lower-level procedures, but changes often occur at the lowest levels. This infers that high-level modules or procedures that should be unaffected by such detailed modifications may be affected due to these dependencies. Again, consider the case where the camera characteristics change, and even though perhaps only one routine needs to be rewritten, the calling module(s) need to be modified and recompiled as well. A preferable situation is to invert these dependencies, such as is evident in the Liskov substitution principle. This principle is at work in the decorator pattern, which is used in the object-oriented case study for the image preprocessing. The intent here is to allow dynamic changes in the preprocessing scheme, which is achieved by ensuring that all the image processing objects conform to the same interface, and are therefore interchangeable.
5.4 Liskov Substitution Principle

Liskov\textsuperscript{25} expressed the principle of substitutivity of sub-classes for their base classes. “What is wanted here is something like the following substitution property: If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \), then \( S \) is a subtype of \( T \).” This principle has led to the concept of type inheritance and is the basis of polymorphism in object-oriented systems, where instances of derived classes can be substituted for each other, provided they fulfill the obligations of a common superclass. Again, the strategies and decorators introduced in the object-oriented case study implement this type conformance and substitutituality, such that any new variations that are desired need merely conform to the supertypes, yet no existing code need be modified.

6 Recommendations and Future Research

The preceding discussions illustrate some of the challenges (in fact one might consider them “habits”) encountered by engineers specifying imaging systems: mixing of operational and descriptive specifications, combining low-level hardware functionality and high-level systems and software functionality in the same functional level, and omission of timing information. It is risky to prescribe a preferred technique because it is well known that there is no silver bullet when it comes to software specification and design, and each system should be considered on its own merits.

Nevertheless, regardless of approach, imaging system modeling should incorporate the following best practices:

- Use of consist modeling approaches and techniques throughout the specification, such as top-down decomposition, structured design, or object orientation.
- Separation of operational specification from descriptive (which ordinarily belongs in a software design document, not a requirements specification).
- Consistent levels of abstraction within models and conformance between levels of refinement across models.
- The modeling of nonfunctional requirements as a part of the specification models, in particular timing properties.
- The omission of hardware and software assignment in the specification (another aspect of design rather than specification).

Finally, this work has sought to describe the appropriateness of object-oriented analysis and design techniques for image processing applications. Through the use of a simple example application, an automated visual inspection system, an object-oriented approach that makes use of the semiformal Unified Modeling Language, has been compared to a process decomposition approach using structured analysis and structured design. At a superficial level, perhaps little advantage can be seen for one over the other, but through the application of some subtle design principles it is apparent that the object-oriented design can better accommodate future changes that are typical in all software systems. In addition, the object-oriented design principles have been codified into a set of reproducible design patterns that make the application of those principles simpler and more efficient, something that does not exist for structured approaches. Finally, as a semiformal methodology, the UML combines the benefits of the more graphical and easy-to-use informal techniques, with all the rigor and benefits of formal methods.

The current focus and future of the research is the application of these patterns in the design and construction of frameworks for image processing systems, both in general and for special purpose image processing subdomains such as compression, multimedia, inspection, or enhancement.

References

18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, New York (1994).
Phillip A. Laplante is an associate professor of software engineering at the Penn State Great Valley School of Graduate Studies. His research interests include real-time and embedded systems, image processing, and software requirements engineering. He has authored numerous papers, 14 books and cofounded the journal *Real-Time Imaging*. He currently edits the CRC Press Series on image processing and is on the editorial board of four journals. He received his BS, MEng, and PhD in computer science, electrical engineering, and computer science, respectively, from Stevens Institute of Technology, and an MBA from the University of Colorado. He is senior member of the IEEE, a member of ACM and SPIE, and a registered professional engineer in Pennsylvania.

Colin J. Neill is an assistant professor of software engineering and the professor in charge of software engineering at the Penn State Great Valley Graduate Center near Philadelphia, Pennsylvania. Prior to joining Penn State, he was a research officer in the Department of Electrical and Electronic Engineering at the University of Wales Swansea, United Kingdom. It was here that he received his BEng in electrical and electronic engineering (1993), MSc in communication systems (1994), and PhD in real-time system design (1997). His research interests include development techniques and processes for software systems, pragmatic formal methods, and applied artificial intelligence.