3 Software Engineering for Image Processing and Analysis

Dietrich Paulus, Joachim Hornegger, and
Heinrich Niemann

Lehrstuhl für Mustererkennung
Universität Erlangen-Nürnberg, Germany

3.1 Introduction

Configuring or programming image processing systems is a time-consuming task which requires specialized knowledge on the effects of image processing algorithms as well as knowledge about the implementation and interfaces. Clearly, software engineering is required for the application programmer of image processing systems. But even people who do not implement their applications themselves occasionally have to face software engineering problems. Several commercial or free packages for image processing exist providing routines which can be plugged together to more complex operations. This does not solve software engineering problems; it rather shifts the basic building blocks to a higher level. The larger the application gets which uses such libraries, the higher is the importance of well-structured software.

The major problem in design of general imaging systems is that on the one hand highly run-time efficient code and low-level access to hardware is required, and that on the other hand a general and platform-independent implementation is desired which provides all data types and functions also for at least intermediate-level processing, such as results of segmentation.

Software re-use is crucial in any large system; well-documented packages should be usable even across applications. If every programmer is allowed to re-program existing code, soon several pieces of code will be scattered around in the system which serve the same purpose.

Today's software engineering is closely coupled with the ideas of object-orientation. Theoretically and practically, object-oriented programming gains high attention. Object-oriented programming can help simplifying code re-use; if applied properly, it unifies

interfaces and simplifies documentation by the hierarchical structure of classes. A key mechanism of object-oriented programming is polymorphism [Cardelli–Wegner 85]. In this article we give examples of how polymorphism can simplify image processing programs and how it can keep the required efficiency.

In this contribution we describe the use of object-oriented programming for the implementor of image processing and analysis software. In CVA III, Sect. 3.9, a similar issue is shown with an emphasis on the configuration of an application system. In CVA III, Sect. 3.12, an alternative to object-oriented programming is shown by the introduction of independent generic modules which can also be used in combination with object-oriented programming.

We start with a short introduction of the general terminology of object-oriented programming in Sect. 3.2. Software developers for image processing today have the choice from several programming languages suited for their needs, e. g. C C++ Java Ada Fortran etc. (Sect. 3.3). In general, image analysis has to use knowledge about the task domain. We outline an object-oriented implementation of a knowledge based image analysis system in Sect. 3.4; In Sect. 3.4 we emphasize implementation issues, whereas in CVA II, Chap. 2 the general structure is described formally. In Sect. 3.5 a novel architecture for classes representing data, actions, and algorithms for image processing is presented. This architecture is applied to segmentation, object recognition, and object localization. We conclude with a summary in Sect. 3.6.

3.2 Object-oriented software engineering

Object-oriented programming has become popular in many fields including imaging applications. We briefly introduce the important ideas and terms of object-oriented software and the basic principles for object-oriented analysis, design, and programming. Especially, we discuss those software engineering issues which are relevant for image processing.

3.2.1 Object-oriented principles, analysis, and design

The object-oriented programming style suggests the decomposition of the problem domain into a hierarchy of classes and a set of communicating objects, which are instances of classes. The object-oriented programmer specifies *what* is done with the objects; the procedural way of programming uses aspects of *how* something gets done. One advantage of object-oriented software design is the one-to-one assignment between concepts in the application domain and the objects in the program. Even the analysis of the problem domain has to be involved in this mapping. Analysis and program design are no longer separated in the software development process; object-oriented analysis and design share the same terminology and tools. The first phase of any software development is to define the requirements. Three other connected stages which are shortly described in the following sections are common to object-oriented software develop-

ment. The most important ideas of object-oriented software which we introduce in the following sections are *objects*, *classes*, *inheritance*, and *polymorphism*.

In the object-oriented analysis (OOA) stage, concepts of the *problem domain* and their correspondence are identified and specified. These *objects* are grouped to *classes*. Hierarchical relations between these classes are used; information which can be shared by several special classes will be included in a general class and passed to the special cases by *inheritance*. Objects are decomposed into their components which are again described as classes.

Example 3.1: Object-oriented analysis

A typical problem domain from the area of image analysis is the recognition and localization of industrial objects on an assembly line. Concepts of this domain are the various parts, the belt, or non-physical terms like speed, motion, or a stable position for the object. A hierarchic order of parts may group the parts according to their purpose. General terms can be identified which are used for a group of concepts.

In the object-oriented design (OOD) phase the attention shifts slightly towards the implementation domain. The conceptual class hierarchy created in OOA is overlayed with links which are meaningful for the implementation only. This causes a transition from the problem domain to the *solution domain*.

Ideally, two hierarchies are used. One relates the classes specific to the application domain which were drafted in the analysis phase. The other hierarchy provides the implementation concepts, like sets, lists, or geometric objects. These two hierarchies are linked together, possibly creating multiple inheritance relations.

So called *methods* are defined for the new classes. These methods provide access to the data represented in the classes and also perform the intended transformations on the objects.

Example 3.2: Object-oriented design

In example 3.1, the classes for the industrial objects can now use geometric object classes to describe their shape, e. g., a wheel will use a circle class. This combines the specific application with general definitions which are independent of the application.

Several graphical representations and mechanisms have been proposed for OOA and OOD in the past. The proposals of Booch, Coad & Yourdon, Jacobson, Rumbaugh et al., and Shlaer & Mellor used similar ideas, each with its own flavor and with different notation. G. Booch, I. Jacobson, and J. Rumbaugh joint their efforts and created the "Unified Modeling Language" (UML) which includes three essential parts [Breu–Hinkel–Hofmann et al. 97]:

- guidelines for the vocabulary,
- fundamental modeling concepts and their semantics,
- notation for the visual rendering of the concepts.

This language has a considerable syntactical complexity and requires advanced programming skills. Nethertheless it gains industrial attention widely, although no publications

are known which use this notation for imaging applications, yet. Since these skills are required for the implementation of image analysis as well, the language is useful for our purpose. We will use it in the following (e. g. in Fig. 3.3) and introduce the very basic notation briefly.

Classes are represented as boxes, divided up to three fields; the upper field contains the class name,[1] the middle field contains data fields called attributes, and the lower field contains method names. An extra small box in the left upper corner marks a template; the actual type is inserted here for a template instantiation. Except for the class name, the fields may be empty. Types and arguments are listed as well, as we will see in the examples below. Classes which are merely used as a common interface definition for further derived classes are called *abstract classes* in the object-oriented terminology; the name is printed in italics in UML.

An arrow with an empty triangle as arrow head relates two classes by inheritance, pointing to the base class. Template instantiations use a dashed line to relate to the template. A line with a filled diamond at the head denotes composition; an empty diamond is used for aggregation. In the following sections we will see several examples.

3.2.2 Object-oriented programming

After analysis and design, object-oriented programming (OOP) can take place. Classes are used for the implementation of actions or tasks, i. e. algorithms, as well as information, i. e., data. As we outline in Sect. 3.4.2, classes can also be used to provide easy and portable access to devices such as frame grabbers, or access to actors which are commonly used in active vision. As shown in CVA III, Chap. 3.12, implementations for image operations can gain run–time efficiency if generic modules are used. Classes can be used to wrap these generic interfaces and to provide a uniform interface by inheritance.

Software re-use is highly desired due to the high costs of programming. Modularity is a central concept which helps maintaining large systems. Data abstraction provides clean interfaces which are essential when several programmers share code in a team. One other goal of software engineering is to provide components with a long lifetime, even when changes are required. These principles have been known since years in the context of object-oriented programming they have now gained attention widely. Object-oriented design cannot guarantee that these principles are fulfilled, but the strong interconnection of OOD, OOA, and OOP simplifies updates and evolution. In contrast to traditional software engineering, these three stages are not strictly sequential; a return from a later stage to a previous one is possible and intended.

The programming language C++ in particular has the advantage that it combines efficient conventional constructs with object-oriented features. Existing routines in C which sacrifice clean structure to gain speed — which unfortunately is necessary in

[1] Instead of the technical name in the syntax of the programming language, we will use a descriptive term in the following figures.

some rare cases for image processing — can be encapsulated in classes or objects which provide safe interfaces.

Generally, methods in object-oriented programming have fewer arguments than corresponding function calls in traditional programming, since parts of the required information may be already bound to the object (for example, a FFT object may have its internal tables for the the actual image size, and no such tables will have to be allocated for the function call and passed to it as arguments). This again facilitates software maintenance and re-use, especially if a class and its interface has to be exchanged. Fewer modifications are then required in the code, compared to conventional programs.

Reuse of general class libraries serves for two purposes. Common programming problems — like the implementation of linked lists or sets — have already been solved in these systems and can be used without further effort. Exchange of software using such class libraries is simplified since the classes share the same structure and interfaces.

The major idea of object-oriented programming now is to define abstract interfaces in general classes, i. e., classes which are in the higher levels of the class inheritance graph, and to provide a specific implementation in the derived classes. If an algorithm now uses only the interfaces available in the more general classes, then the outcome of the process depends on the actual object to which the method is applied. The type of this object may vary and can be taken from several derived classes. This behavior is called *polymorphism.*

3.2.3 Software engineering for image processing

Real-time constraints, efficiency, and the large amount of data impose special software problems to image processing systems. The basic requirements for designing a general software system for image processing (in the sense of the invariant part of a system in CVA III, Sect. 3.7) are:

1. Access to imaging hardware has to be possible; this includes capturing devices, graphics, etc. as well as camera interfaces, camera motors, etc. Since hardware development cycles are much faster than software change, and since software re-use is desired, the interfaces have to be encapsulated by portable definitions.
2. Naturally, highly efficient — yet safe — access to vectors and matrices has to be possible.
3. Input and output has to be fast, efficient, and machine-independent. This has to be guaranteed not only for low-level data structures such as image matrices, but also for intermediate– and high-level data such as segmentation results and knowledge bases, as well.
4. Image processing and analysis modules should be as independent as possible from the final application, in order to be re-usable accross systems.

The discussion of object-oriented programming for image processing started when C++ became known. This programming language promised to provide the efficiency required for image processing, combined with object-oriented programming, and the possible code re-use by the upward compatibility to C.

The test phase is a crucial part of software design and programming. Systematically organized tests will increase the reliability of software for real-world applications. Problems not considered during the analysis might be detected during tests. A general problem is that tests can reveal only the existence of bugs, but generally cannot be used to prove that there are no errors in a program. The more you test, however, the lower will be the chance that there is a hidden bug in your code. Some general guidelines will help you testing your software:[2]

1. Put assertions in every method and function.[3] Test the module separately and undefine these assertions when either run-time is crucial for further tests, or when you are sure the assertions never fail.
2. Each branch in the program should be activated at least once during the test. This comprises the parts for rare error conditions as well!
3. Every loop should be used at least twice during a test suite.
4. Try to imagine irregular, unexpected, wrong, inconsistent input and test your routines with such data, for example:
   (a) images of size 1×10000 or even of size 0×0,
   (b) images with constant intensity value at every pixel,
   (c) sequences of zero length.
5. Use more than one image for testing. Vary all possible parameters, such as intensity, size, number of objects, contrast, etc.
6. Use at least one data set as input for which you know the expected output of the program.
7. Test predictable special cases, such as discontinuities of functions, division by numbers close to zero, etc.
8. Keep in mind the limitations of resources, such as storage or execution time. Estimate the required resources. Verify that predictable behavior of your program is guaranteed even if you exceed the limits.
9. Verify that users will accept your software and use it for their needs.

3.3 Programming languages for image processing

We survey existing programming languages with respect to their usefulness for image processing and image analysis.

[2] Collected from the world wide web, from lecture notes, and from personal experience.
[3] Most C and C++ environments provide an efficient and effective solution by a simple `assert` macro.

3.3.1 Conventional Programming Languages

Early image processing systems were mostly written in Fortran, e. g. SPIDER [Tamura 83]. Although this language still has its users, mostly because if its mathematical capabilities and libraries, only few image processing applications written in Fortran remain.

Since the ideas of IKS [Gemmar–Hofele 90], several new software architectures for image processing have been proposed, finally resulting in an international standard PIKS [Blum–Hofmann–Kromker 91, Butler–Krolak 91, Standard 94, Pratt 95]. This standard contains an interface to Fortran programs.

A software architecture was planned as a common basis for program exchange between companies, research institutes, and universities is presented in [Kawai–Okazaki–Tanaka–Tamura 92]. This system is now written in C++ and extends the ideas of SPIDER [Tamura 83] and offers more than 500 algorithms for image processing.

For image processing, at least one decade was dominated by the use of C (e. g. [Dobie–Lewis 91, Parker 97]). On the one hand, this language lacks most of the higher mathematical notation; on the other hand it provides efficient access to low level bit manipulations. The latter is very useful for low level image manipulation, such as masking out areas in an image. The missing mathematical definitions in the language *syntax* are compensated by the large number of mathematical libraries which are available for the language. Rather than operators of the language, function calls to the libraries have to be used to apply mathematics.

The Khoros system [Rasure–Young 92, Young–Argiro–Kubica 95] is an environment for interactive development of image processing algorithms. The system includes a neat visual programming environment. The algorithms can also be run without interactive graphics. The system provides a large C-language library of imaging functions (over 500 functions), some of them used in $2\frac{1}{2}$D and 3-D image processing. Knowledge based processing is not part of this package.

The PIKS standard mentioned above is also specified for the C language.

3.3.2 Object-oriented programming

Object-oriented programming languages have been known to computer scientists since over 25 years. The ideas originated in the ancestors Simula and Smalltalk. During this period of time, the (conventional) programming language C had its breakthrough in the world.

In the late eighties, the C language was extended with object-oriented ideas. The language C++ [Stroustrup 91] mainly used the ideas of Simula. C++ is almost a superset of C; i. e., most C programs are C++ programs as well. Many valuable image processing routines nowadays written in C can be re-used without modification. Possibly because of the cheap or free availability of C++-compilers — even on personal computers — this language had enormous success.

Several new object-oriented languages have been invented, such as Eiffel, CLOS, In the following, we report only on those languages which are currently relevant in imaging applications.

The programming language C++ [Stroustrup 91] had an overwhelming success in the last few years in all application areas. Many image processing projects started in C and are now extended or converted to C++, e. g. [Eckstein–Lohmann–Meyer–Gruhl et al. 93, Young–Argiro–Kubica 95]. Although the C++-language is only partially suited for object-oriented programming, it presently seems to be the best choice to combine efficiency with object-oriented design in real world applications, especially those operating under real-time conditions. For a discussion of image processing systems see e. g. [Carlsen–Haaks 91, Coggins 87, Zimmer–Bonz 96, Paulus–Hornegger 97, Piper–Rutovitz 88]

One major disadvantage of C++ was its lack of standard general purpose classes, such as linked lists. Meanwhile, the *Standard Template Library* (STL) [Musser–Saini 96] has become part of the C++ standard and is distributed with most compiler. Since it uses templates rather than inheritance, it gains run-time efficiency at some points compared to object-oriented class libraries. STL supports genericity (CVA III, Chap. 3.12); this is a somewhat orthogonal idea to object-oriented programming which is based on inheritance. In C++, polymorphism is supported at run-time by `virtual` functions; these functions are internally called indirectly via a pointer, rather than by a direct function call. Template instantiation at compile-time provides a variant which is called *compile-time polymorphism.*

General purpose classes are also available for C++ as class libraries. A system called NIHCL [Gorlen–Orlow–Plexico 90] incorporates many Smalltalk ideas into C++ and uses inheritance rather than templates. In contrast to STL, NIHCL provides storage and recovery of arbitrary objects in a unified way, as it was possible in Smalltalk. Our image analysis system described in Sect. 3.5 uses this general library.

Since several years, people work on the image understanding environment; examples can be found in [Haralick–Ramesh 92] and in various articles in the proceedings of the *Image Understanding Workshop*, e. g. [Mundy–Binford–Boult et al. 92]. This large system is implemented in C++ as well and partially uses STL.

The machine vision system described in [Caelli–Bischof 97] is implemented in C++ and uses class hierarchies for the knowledge base as well as for the interfaces to image processing algorithms.

The programming language Java [Lyon–Rao 97] promises a new era in programming[4]. It is defined as a portable language where the portability extends down to the level of binary programs. This is achieved by the so called "Java virtual machine" which *interprets* the compiled programs. This interpreter has to be present on the host machine or hardware support has to be provided. Java is an object-oriented language with a clean and simple syntax definition, much simpler than C++ although with a similar outlook. The language is type safe which helps creating reliable programs. The

[4] e. g. http://rsb.info.nih.gov/ij/

language has no pointer variables which is somewhat unusual. Many deficiencies of C++ have been cured in this new language. The compilers come with a rich library of general purpose classes which greatly simplifies sharing code with others.

Although first highly efficient implementations exist and applications to signal processing have been proposed [Lyon–Rao 97], Java currently lacks the required run-time efficiency for image processing applications.

In its original definition, the programming language ADA was an object-based language; it was not an object-oriented programming language since it did not support inheritance which is an essential object-oriented feature. Because of the high complexity of the formal syntax, it took several years before complete compilers were available on any platform. Although the language provides efficient, portable libraries, and concepts for software re-use, it was not accepted by the image processing population. Instead, the language C was used all over the world.

The recent re-definition in [Barnes–Brosgol–others 95] added the missing object-oriented features. The syntax complexity remained, however.

Smalltalk is *the* prototype of the object-oriented paradigm. Since it is an interpreted language it is generally slower than a compiled program. It is also not particularly suited for mathematical problems. Smalltalk can be used to program the higher levels in image understanding; since the language has a large number of classes for various general problems, solutions can be formulated elegantly. When it comes down to pixel access for image preprocessing, the language is not the right choice.

3.3.3 Proprietary image processing programming languages

Some systems use an own programming language for image processing, which is usually either interpreted from a textual description or a graphical user interface. The Cantata language in Khoros [Rasure–Young 92, Young–Argiro–Kubica 95] is one typical example. Heurisco [Jähne 97] uses a textual as well as a graphical interface. Ad Oculus [Bässmann–Besslich 95] has a graphical description. The major advantage is that usually only few keystrokes are required for even complex imaging operations. The drawback is that a new language has to be learned and that this language is not portable to other systems, unless the interpreter exists on the target platform. One special case of this idea are image algebras where image processing is formalized as an algebraic problem, e. g. in [Giardina 84].

If image processing is seen as a data flow problem (as in Sect. 3.4.1), the command interpreter of the operating can be used to compose operations from single processes. This strategy has been applied in the system HIPS [Landy 93] which uses Unix-pipes to create sequences of operations. The Khoros system also provides a command line interface; the automatically created sequences of program invocations use intermediate files as interfaces.

The system presented in [Cracknell–Downton–Du 97] uses the Tcl/Tk language and interface to compose image understanding algorithms. The components may be written in C or C++.

3.3.4 Summary

In Sect. 3.2 we argued that object-oriented ideas greatly help reducing difficulties in software for image processing. In Table 3.1 we summarize those features of object-oriented programming languages which are of interest for imaging. Parallel processing is of course useful for image processing, especially when processing times are crucial. Although we did not treat this subject in the previous sections, we list in Table 3.1 whether concurrent and distributed processing is supported by the syntax of the programming language.

Although ADA'95 provides object-oriented features, it is barely used in scientific projects, except in those related to space technology; the programming languages C, C++, and Java are currently used in image processing implementations. In the table we note whether the language is currently used for image processing and image analysis (ip/ia). We also have a column which shows whether the language is type-safe; this feature, if provided, makes the tests performed by the compiler more reliable and thus increases software stability.

Table 3.1: *Features of object-oriented programming languages. Extended from the table in [Goedicke 97].*

	type-safe	syntax complexity	ip/ia	concurrency tools
C++	low	high	yes	not part of the language
Java	yes	low	no	primitives exist
Ada	almost	high	few	fully suppored
Smalltalk	yes	low	no	coroutines

Although most programming languages allow mixtures with subroutines of other languages, we assume that the design goal in most cases is to have a uniform architecture as well as a uniform programming language. C function calls in C++ can be seen as almost conformant with this guideline and are acceptable, since the benefit of reusing existing C code is much higher than the penalty of some minor extra interface requirements.

The language Objective-C encapsulated Smalltalk features in the C language. The system described in [Carlsen–Haaks 91] is one of the few references to applications of Objective-C in imaging applications, combined with object-oriented extensions to LISP and Prolog. Today's best choice for a programming language for image processing and understanding is C++.

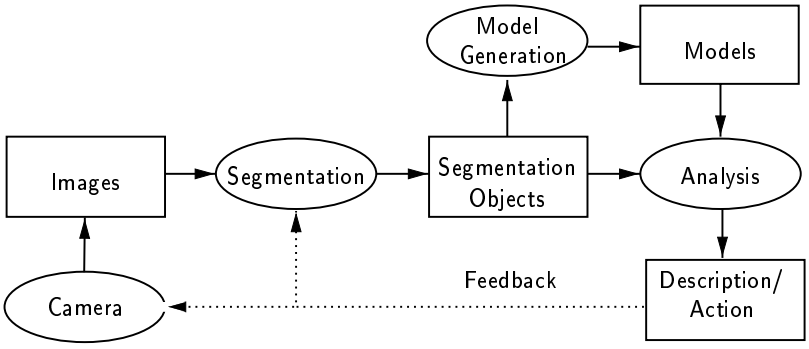

Figure 3.1: *Data flow in an image analysis system.*

3.4 Image understanding

In this section we describe the general software architecture of image understanding (IU) systems and apply object-oriented principles.

3.4.1 Data flow

The general problem of image analysis is to find the best description of the input image data which is appropriate to the current problem. Sometimes this means that the most precise description has to be found, in other cases a less exact result which can computed quicker will be sufficient. This task may be divided into several sub-problems. After an initial preprocessing stage, images are usually segmented into meaningful parts. Various segmentation algorithms create so called segmentation objects [Paulus–Niemann 92].

Segmentation objects are matched with models in a knowledge base which contains expectations of the possible scenes in the problem domain.

The various data types involved in image segmentation, like images, lines or regions, may serve for data abstraction of the problem. In object-oriented programming, these data types are naturally represented in classes. Segmentation may be seen as a data flow problem relating these various representations. An overview of the main components is shown in Fig. 3.1; data is captured and digitized from a camera and transformed to a symbolic description or results in an action of the system. Image processing tasks are shown in oval boxes; data is depicted as rectangles; in the object-oriented design phase, both will naturally be grouped to class hierarchies. If required, the algorithms can be implemented as generic modules for differnt types, as shown in CVA III, Sect. 3.11.

The problem of finding an optimal match and the best segmentation can be seen as an optimization problem and is formulated as such in CVA II, Chap. 2. Optimization may search for the best possible match as well as include efficiency considerations which are crucial for real-time image processing.

Knowledge based vision uses a model of the scene for image interpretation. The scene may be decomposed into object-models which can be represented using their structural properties and relations (e. g. in a semantic network CVA II, Sect. 2.2.1), or as statistical object models ([Hornegger–Niemann 94], CVA II, Sect. 3.5.3).

The system architecture in Fig. 3.1 implies various interactions between modules. Information has to be exchanged by means of well defined interfaces. Modules and data structures with well defined interfaces are naturally implemented as classes in OOP or as modules in structured programming.

The segmentation object is a central idea for the data representation independent of the algorithms used for image segmentation, and can be used as an interface between data-driven segmentation and knowledge-based analysis. Models can be described in a similar formalism [Niemann 90].

3.4.2 Devices and actors

The output of the system in Fig. 3.1 is a description of the input image data. In active vision systems (Chap. 3.10), the output may additionally or alternatively contain control commands for the sensor device or for the actor (e. g. a robot or a moving vehicle). This provides feedback from high-level processing to data driven segmentation, or even to the image capturing devices (dashed line in Fig. 3.1). This data flow is also common to active vision systems, the parameters of the visual system are adjusted by the controlling computer in order to get the best possible image for the actual analysis purpose. The system design in Fig. 3.1 is therefore suitable for conventional image analysis as well as for active vision.

Interaction with graphical and pointing devices is necessary for interactive computer vision systems, as they are common e. g. in medicine. Interaction with physically moving tools requires a control loop which can be closed in real-time.

Naturally, these actions and devices are modelled as classes and objects in OOA and OOD. Their well defined interface facilitates data exchange between programs and devices.

Example 3.3: OOD for a frame grabber

Image processing software has to access imaging hardware, e. g., to a frame grabber. Most systems provide libraries which are to some extent portable to the next generation of the same hardware. Although the functionality of two frame grabbers of different manufacturer may be rather similar, the software interface will be of course different.

Instead of the traditional approach which scattered `#ifdef`'s around in the code to be able to compile a program for different interfaces, the object-oriented designer creates a class which encapsulates the common features of several hardware interfaces, for example, for setting the resolution of an input image or for selecting the input connector. The algorithms use this general — polymorphic — interface and are independent of the underlying hardware.

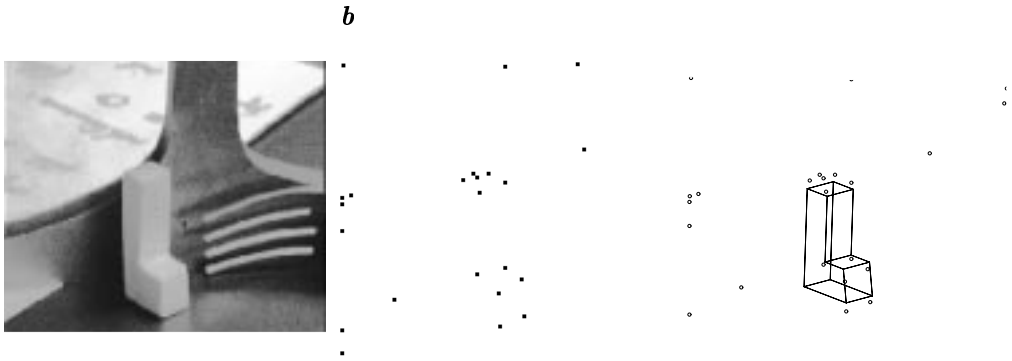

Figure 3.2: *Example of a scene with heterogeneous background:* **a** *gray-level image,* **b** *segmentation result, and* **c** *estimated pose*

3.4.3 Statistical object recognition

In a Bayesian framework for object recognition using 2-D images ([Hornegger 96], CVA II, Sect. 1.4), statistical model generation, classification, and localization is based on projected feature vectors. Localization is expressed by the rotation and projection matrix and translation vector. The objects are taken from a set of possible classes $\kappa \in \{1 \cdots K\}$ and described by parameter vectors, matrices, or sets.

Object localization and recognition corresponds to a general maximization problem CVA II, (1.4).

Fig. 3.2 shows a gray-level image (a) and the resulting set of 2-D point features (b), if a standard corner detection algorithm is applied. Fig. 3.2c shows the result of the maximization of CVA II, (1.4). Similar results are shown in CVA II, Fig. 1.7.

In order to implement such an optimization, the function to be optimized has to be independently specified from the algorithm performing the optimization. Whereas this is written down easily in mathematics, it requires clean design in the implementation taking into account the computational complexity of the optimization. We outline a solution in Section 3.5.3.

3.5 Class hierarchy for data and algorithms

Whereas hierarchical data representation by classes has become almost state of the art, hierarchies of classes for operations, algorithms, and actors are not common, yet.

We first describe a hierarchy of classes which facilitate simple interfaces for image processing; image processing operations are implemented as another class hierarchy which uses the data interfaces [Paulus–Hornegger 97]. A third hierarchy provides various optimization algorithms. These ideas are realized in **An im**age **a**nalysis **s**ystem which is written in C++.

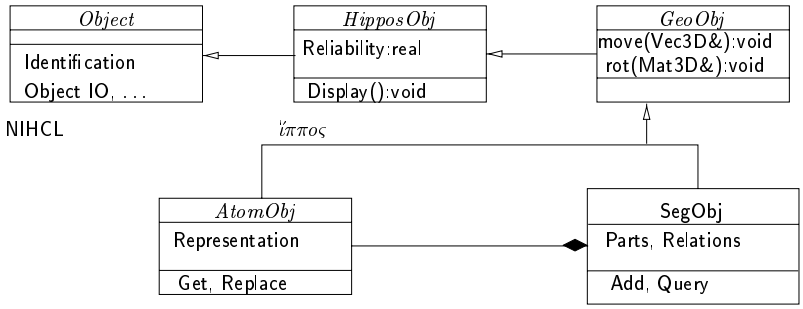

Figure 3.3: *A section of a hierarchy of geometric objects for segmentation (in total approximately 150 classes).*

3.5.1 Data representation

Various data representation schemes have been developed for data in image analysis covering image data structures as well as results from segmentation. Some of them may be treated as algebras with more or less complete sets of operations (e. g., chain codes or quad trees). Other representations are used because of their storage efficiency (e. g., run length codes), others because of their runtime efficiency. Such ideas were combined into a **Hi**erarchy of **P**icture **P**rocessing **O**bject**S** (HIPPOS, written as *ίππος* [Paulus–Niemann 92]).

A central problem visible in Fig. 3.1 is the data exchange of segmentation results, which are an initial symbolic description of the image. The solution in *ίππος* is a hierarchy of classes, which is shown in Fig. 3.3. The general classes for implementation which are added in OOA (Sect. 3.2.1) were taken from the NIHCL class library [Gorlen–Orlow–Plexico 90]. All segmentation results can be stored in an object of class `SegObj` no matter whether the data is computed from line-based or region-based segmentation, color, range, or gray level images. This object representation can be exchanged between different processes which possibly run on different computer architectures. This general tool is used in many algorithms. A segmentation object is a geometric object (`GeoObj`). It consists of a set of parts which are in turn geometric objects. Atomic objects are geometric objects as well and end the recursion. The abstract base class `HipposObj` connects to the NIHCL object and serves as a common root for all image processing classes. It contains a judgment attribute which is used for the comparison of competing segmentation results by the analysis control (cmp. CVA II, Sect. 2.6).

As our applications, e. g., in [Denzler–Paulus 94, Beß–Paulus–Niemann 96] show, this has proven adequate for 2-D, $2\frac{1}{2}$D and 3-D image analysis. Other sub-trees exists for image classes, like gray level images, stereo images, range images, color images, etc. as well as classes for lines, circular arcs, polygons, chain codes, regions, active contours etc. The whole *ίππος*-hierarchy currently consists of approximately 150 classes.

Vectors and matrices are implemented as templates using the concept of genericity (CVA III, Sect. 3.12). In order to be efficient, pixel access to images should not be programmed via virtual functions. A flexible, safe, and simple mechanism is used instead which is shown in example 3.4. Pixel access by operator syntax is possible for vectors as well as for matrices without loss of speed [Paulus–Hornegger 97]. The index operator in example 3.4 can do a check on the validity of the operand thereby eliminating most of the common errors in image processing (which cannot be detected in C automatically).

Example 3.4: Efficient and safe pixel access

```
template <class T> struct Vector { // simplified version
  T * array; int size;             // the actual data
public:
  T& operator[] (int i)
{ /* check index validity here */ return array[i]; }
  Vector(int);              // allocate internal pointer
  Vector(int,T* p);         // use already existent storage in p
};
template <class T> struct Matrix { // simplified version
  T ** tarray; int size;  // contiguous allocation for the matrix
  Vector<T>** varray;      // the varrays internally use tarray
public:
  Matrix(int,int);        // will allocate C-compatible tarray and
  // use second construtor for vectors
  // using this array
  operator T**() { return tarray; }  // provides C compatiblity
  Vector<T>& operator[] (int i)
{ /* check index here */ return *varray[i]; }
};

void t()
{
   Matrix<int> m(10,10);  // define a matrix object
   m[3][4] = 3;           // safe pixel access
   int** mp = m;          // convert to pointer access
   mp[3][4] = 4;          // fast pixel access
}
```

The idea in example 3.4 can be used to implement sub-images [Paulus–Hornegger 97]. An interface to commonly available C-functions is easy by the automatic conversion operator to type `T**`; Since these generice modules are integrated into the NIHCL class hierarchy, they share methods for input and output of objects.

3.5.2 Image operator hierarchy

Many operations in image processing can be structured hierarchically in a straight forward manner. Such hierarchies can be implemented in a hierarchy of classes for

operations in a straight forward way (cmp. [Carlsen–Haaks 91, Faasch 87]). Objects are the actual algorithms with specific parameter sets which are also objects [Harbeck 96]. Classes as implementation of algorithms are particularly useful, when operations require internal tables which increase their efficiency. The requirement stated in CVA III, Sect. 3.8 that algorithms should provide meta–knowledge about themselves, can be fulfilled by operator classes; provide polymorphic methods list the type of the operator, the required arguments, and further administrational information. Programmers can be forced by the compiler to implement these functions.

The major advantages of operator-classes are threefold.

- Algorithms can be programmed in an abstract level referencing only the general class of operations to be performed; extensions of the system by a new derived special operator will not require changes in the abstract algorithm.
- Such an extension cannot change the interface which is fixed by the definition in the abstract base class. This guarantees uniform and thus easy-to-use interfaces.
- Dynamic information about the operator, which is actually used, is available. For example, a program may just reference a filter object; during run time it will be decided which concrete filter should be used. Using virtual functions in C++, the run time overhead is negligible Sect. 3.5.5.

The segmentation in [Harbeck 96] accepts as input an image object. An edge-operator-object such as a Sobel-object or a Nevatia-object converts this input image to an edge image. Edges are linked to line objects collected in a segmentation object. Corners are detected on the line and added to the segmentation object with one of several corner detector objects. Later, vertices are identified and the lines are approximated by circular arcs and straight lines using a split-and-merge object. This sequence of operators introduces a refinement of the data flow in Fig. 3.1. Figure 3.4 shows the hierarchy for line segmentation. The segmentation result in Fig. 3.2 as well as the results in CVA II, Fig. 1.7 are computed using this sequence; both point sets are collected in an object of class `SegObj`. Parameter blocks can be shared between different operators as parameter objects. Function call C++ syntax for these operators as shown in example 3.5 is used in Animals which facilitates migration of existing conventional programs to operator objects.

Example 3.5: Image Operator Classes in C++

```
class Sobel: public EdgeDet {     // Sobel operator as a special case
public:                           // of an edge detector
   static const int maxStrength;  //  = 2040;
   virtual void operator() (const GrayLevelImage&,EdgeImage&) ;
};
class Prewitt : public EdgeDet {  // Prewitt edge operator
public:
   static const int maxStrength;  //  = 1020;
   virtual void operator() (const GrayLevelImage&,EdgeImage&) ;
};
```

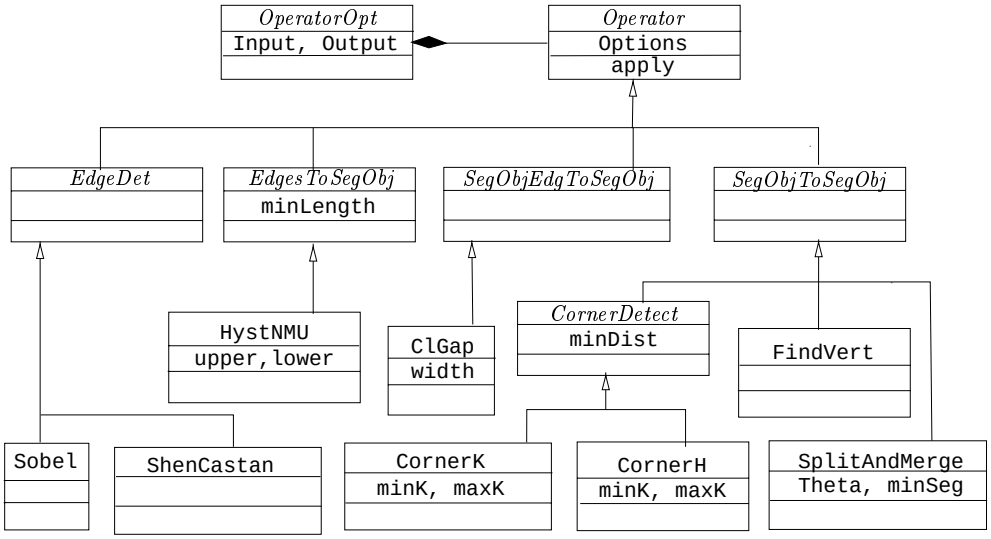

Figure 3.4: *Subtree of the operator hierarchy for image segmentation; from [Harbeck 96].*

Whereas *one* object of a Sobel object will usually be sufficient in a program, several objects of a certain operator object will be needed if these objects keep their internal state and auxiliary data between invocation. As an example consider a discrete Fourier transform which uses tables for sine and cosine values; the size and contents of these tables varies however upon the frame size to be transformed. Several Fourier-transform objects may thus be used in one program for transformations of images, e. g. in a resolution hierarchy. In contrast to conventional solutions, this requires neither code duplication nor complicated management of functions with local storage. In example 3.6, a simple interface to an FFT object is shown. Each instance of the FFT class has an associated internal size and can be used to transform a vector of this size. Again, function call syntax is used.

Example 3.6: Implementation of class FFT

```
class FFT : public IP_OP {
   Vector<double> sintab, costab;  // internal tables
public:
   FFT(int s) : sintab(s), costab(s) { /* init tables here */ }
   Vector<complex> operator() (const Vector<double>&);  // apply FFT
};

void f()
{
  FFT f128(128);  // init internal tab for 128 entries
  FFT f256(256);  // init internal tab for 256 entries
```

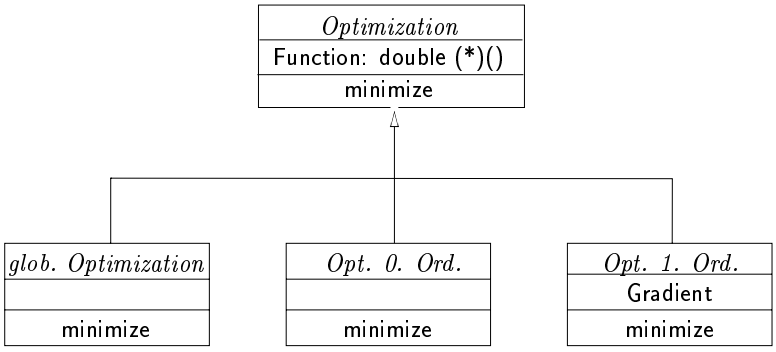

Figure 3.5: *Partial view on class hierarchy for optimization algorithms.*

```
  Vector<double>  v128(128), v256(256); // input data
  Vector<complex> c128 = f128(v128);    // resulting spectrum 1
  Vector<complex> c256 = f256(v256);    // resulting spectrum 2
}
```

3.5.3 Hierarchy for optimization algorithms

The optimization problem in CVA II, (1.4) and in CVA II, example 1.6 requires that several strategies for optimization are evaluated in order to find efficient object recognition strategies. Probabilistic optimization routines which allow practically efficient solutions are discussed in [Hornegger 96]. Again, a class hierarchy for optimization strategies similar to the operator hierarchy above simplifies the experiments.

The basic idea of the implementation is to program the algorithms independently from the function which is to be optimized. An abstract base for all optimization strategies has an internal variable which is the function to be optimized; the class provides a method for minimization or maximization to all derived classes.

All optimization algorithms can be divided into global and local procedures; additional information may be present such as e. g. the gradient of the function (Fig. 3.5). Procedures which use the function directly are e. g. the combinatorial optimization, the simplex algorithm, or the continuous variant of the simulated annealing algorithm (Fig. 3.7). The gradient vector can be used for the optimization of first order. Examples are the iterative algorithms implemented in [Hornegger 96], the algorithm of Fletcher und Powell, and the well known Newton-Raphson iteration which are shown in Fig. 3.6. Fig. 3.8 finally shows an overview of other algorithms implemented in [Hornegger 96].

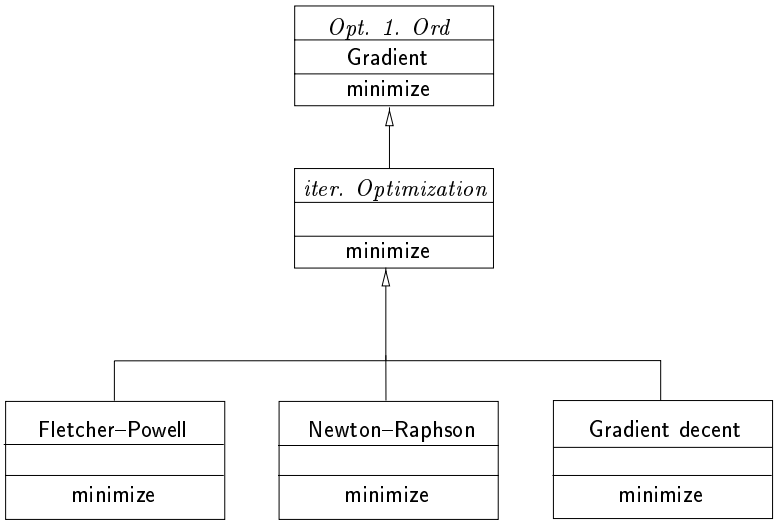

Figure 3.6: *Partial view on class hierarchy for local optimization algorithms of first order.*

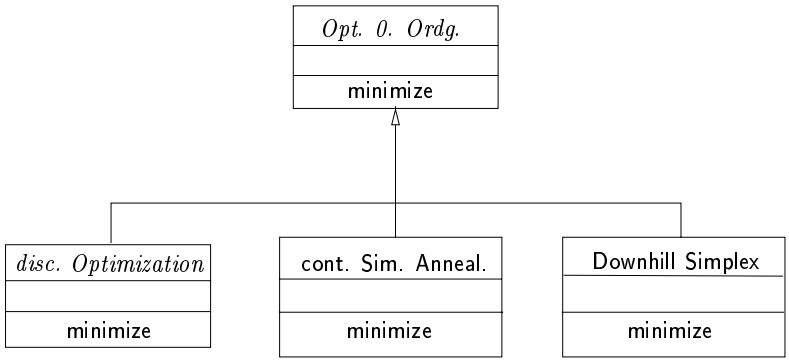

Figure 3.7: *Partial view on class hierarchy for for model densities.*

3.5.4 Hierarchy for actors and devices

A similar technique as in example 3.5 can be used to fulfill requirement 1 on page 67. This is shown in example 3.7 for the case of a camera which can capture either gray-level images or color images and a pan/tilt unit which is commonly used in active vision systems. Of course, a more elaborate internal structure of the `class SMotor` is required in real applications. In fact, a hierarchy of classes providing different interfaces to various hardware is used. This is invisible for the application programmer.

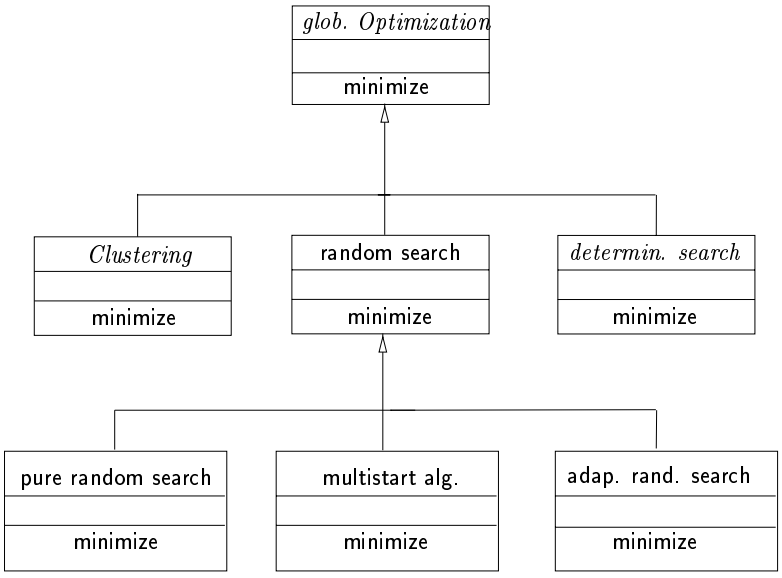

Figure 3.8: *Partial view on class hierarchy for global optimization algorithms.*

Example 3.7: Actor Classes in C++

```
struct SMotor {      // assume stepper motor
   operator=(int p); // will set motor to position p
   operator++();     // will move by one step
};
struct PTVDev : public IP_DEV {
   SMotor panaxis;
   SMotor tiltaxis;
};
struct Camera : public IP_DEV {
   SMotor zoom;                                  // zoom motor is stepper motor
   virtual void operator() (GrayLevelImage&); // will capture the correct
   virtual void operator() (ColorImage&);     // type and size
};
```

Most common functions of such devices can be accessed by operator-syntax. This even releases the programmer from the need of remembering method names since the value for the stepper motor can simply be assigned to the object. The challenging idea behind this concept is not only to encapsulate the hardware interface for each device, but also to provide common interfaces for parts of the device, such as motors.

Table 3.2: *Run time measurements for HP (735/99 MHz), SGI O2 (R10000, 195 MHz), and Dual-Pentium (Linux 300 MHz); all times in milliseconds*

	Number of calls	HP	O2	PC
Direct function call	10^7	199	93	43
Direct method call	10^7	233	93	55
Virtual method call	10^7	634	98	65
Sobel function 256^2 image	10^2	1085	287	300
Sobel object 256^2 image	10^2	1096	290	301
Sobel function 32^2 image	10^4	1551	418	415
Sobel object 32^2 image	10^4	1557	419	175
Safe pixel access	10^7	446	246	84
Fast pixel access	10^7	162	72	37

3.5.5 Efficiency

As shown in Table 3.2, time overhead for virtual function calls is negligible compared to the time needed for the actual computation, such as in the case of the Sobel operator. Of course, trivial computations or pixel access should not be performed using virtual functions, since the ratio of overhead and required processing time for the task is worse, then. The implementation of the Sobel operator uses the fast pixel access via pointers (Table 3.2).

In [Denzler 97], a real-time active vision system was implemented for tracking objects and moving a pan/tilt camera. This system used the image and matrix classes as well as the camera and actor classes in Sect. 3.5 The computation times were small enough to process images and control the camera.

The computation times measured in [Hornegger 96] for the optimization classes outlined in Sect. 3.5.3 were around one minute for 10000 calls to the function to be optimized.

The system has been used in many other applications as well. These references and the measures in Table 3.2 prove that object-oriented programming can be successfully applied in image processing: low–level tasks on matrices and vectors use templates; intermediate results and high–level processing rely on class hierarchies.

The benefit of using object-oriented programming instead of conventional programming is hard to be measured. One measure for the quality of a software system is the duration of its use which which is long if the architecture is accepted by the implementors and which is shorter if the system fails to meet the requirements of the programmers. Our system has been started in 1988 and the oldest class still in use is the one for chain codes which was released in 1989. Our personal experience strongly suggests object-oriented principles for system design in image analysis.

3.5.6 Summary

In Sect. 3.2.3 we listed three essentials for imaging software design. In Sect. 3.5.5 we showed how efficient processing is possible simultaneously with object-oriented design. Several examples were given in the previous sections on access to hardware and devices. The NIHCL object which is used as base class for all classes shown above, provides machine independent persistent objects. This data representation is available not only for images, segmentation objects, etc., but also for actors such as a camera; in this case, the camera parameter settings are stored and can easily be retrieved later.

Of course, object-oriented programming implies that the complete class hierarchy has to be available for the application programmerer, e. g. as a shared library. In contrast, the independent building blocks proposed in CVA III, Sect. 3.12 require almost no common libraries, but share the code for the generic modules. A combination of both approaches will give the best results for complete imaging systems.

3.6 Conclusion

We argued for a clean and uniform software design in imaging software systems. At the same time, a highly efficient implementation is crucial. Only C++ can currently provide these features. For regular data structures such as matrices and vectors, genericity is the right choice; in C++ this concept is available as templates. Starting with images which use such matrices, and ending with complex data representation as segmentation results, classes and inheritance provide clean yet efficient interfaces.

In addition to data, algorithms are grouped hierarchically as classes. While not imposing measurable overhead on the run-time, this idea adds a lot to keep interface problems small and to force programmers to adhere to standards.

The use of classes for interfaces to hardware is to guarantee platform independent access. Hierarchies of such classes facilitate uniform syntax even for varying devices.

A wise choice of the available tools will keep the run-time overhead at a negligible minimum and will on the same time greatly increase the value of the software in terms of reusability and expected maintenance effort.

Acknowledgement

The authors want to thank J. Meixner for his remarks on Sect. 3.2. We acknowledge the software contribution of our various colleagues. Numerous students helped implementing the system. This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG), grant Ho 1791/2-1.

Bibliography

[Arps–Pratt 92]
R. B. Arps, W. K. Pratt (Hrsg.): *Image Processing and Interchange: Implementation and Systems*, SPIE, Proceedings 1659, San Jose, CA, 1992.

[Barnes–Brosgol–others 95]
J. Barnes, B. Brosgol, others: *Ada 95 Rationale. The Language. The Standard Libraries*, Itemetrics Inc., 733 Concord Av, Cambridge, MA 02138, Jan. 1995.

[Bässmann–Besslich 95]
H. Bässmann, P. Besslich: *Ad Oculos*, in *Ad Oculos*, Internat. Thomson Publ., London, 1995.

[Beß–Paulus–Niemann 96]
R. Beß, D. Paulus, H. Niemann: *3D Recovery Using Calibrated Active Camera*, in *Proceedings of the International Conference on Image Processing (ICIP)*, Bd. 2, IEEE Computer Society Press, Lausanne, Schweiz, September 1996.

[Blum–Hofmann–Kromker 91]
C. Blum, G. R. Hofmann, D. Kromker: *Requirements for the first international imaging standard*, *IEEE Computer Graphics and Applications*, Bd. 11, Nr. 2, Marz 1991, S. 61–70.

[Breu–Hinkel–Hofmann et al. 97]
R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, V. Thurner: *Towards a Formalization of the Unified Modeling Language*, in M. Aksit, S. Matsuoka (Hrsg.): *ECOOP'97—Object-Oriented Programming,* 11^{th} *European Conference*, Bd. 1241 von *Lecture Notes in Computer Science*, Springer, Jyväskylä, Finland, 9–13 Juni 1997, S. 344–366.

[Butler–Krolak 91]
T. Butler, P. Krolak: *An overview of the Programmer's Imaging Kernel (PIK) proposed standard*, *Computers and Graphics*, Bd. 15, Nr. 4, 1991, S. 465–472.

[Caelli–Bischof 97]
T. Caelli, W. Bischof: *Machine Learning and Image Interpretation*, Plenum, 1997.

[Cardelli–Wegner 85]

L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*, *Computer Surveys*, Bd. 17, Nr. 4, 1985, S. 471–522.

[Carlsen–Haaks 91]
I. C. Carlsen, D. Haaks: *IKS^{PFH} — Concept and implementation of an object–oriented framework for image processing*, *Computers and Graphics*, Bd. 15, Nr. 4, 1991, S. 473–482.

[Coggins 87]
J. M. Coggins: *Integrated Class Structures for Image Pattern Recognition and Computer Graphics*, in K. Gorlen (Hrsg.): *Proceedings of the USENIX C++ Workshop*, Santa Fe, NM, 9.-10. November 1987, S. 240–245.

[Cracknell–Downton–Du 97]
C. Cracknell, A. C. Downton, L. Du: *TABS, a new software framework for image processing, analysis, and understanding*, in *Image Processing and its Applications*, Dublin, 1997, S. 366–370.

[Denzler–Paulus 94]
J. Denzler, D. Paulus: *Active Motion Detection and Object Tracking*, in *First International Conference on Image Processing*, 1994.

[Denzler 97]
D. Denzler: *Aktives Sehen zur Echtzeitobjektverfolgung*, Infix, Aachen, 1997.

[Dobie–Lewis 91]
M. R. Dobie, P. H. Lewis: *Data Structures for Image Processing in C*, *Pattern Recognition Letters*, Bd. 12, 1991, S. 457–466.

[Eckstein–Lohmann–Meyer–Gruhl et al. 93]
W. Eckstein, G. Lohmann, U. Meyer–Gruhl, R. Riemer, L. A. Robler, J. Wunderwald: *Benutzerfreundliche Bildanalyse mit HORUS: Architektur und Konzepte*, in S. J. Pöppl, H. Handels (Hrsg.): *Mustererkennung 1993*, Springer, Berlin, 1993, S. 332—339.

[Faasch 87]
H. Faasch: *Konzeption und Implementation einer objektorientierten Experimentierumgebung für die Bildfolgenauswertung in ADA*, Phd thesis, Universität Hamburg, Hamburg, 1987.

[Gemmar–Hofele 90]
P. Gemmar, G. Hofele: *An Object–Oriented Approach for an Iconic Kernel System IKS*, in *Proceedings of the* 10^{th} *International Conference on Pattern Recognition (ICPR)*, Bd. 2, IEEE Computer Society Press, Atlantic City, 1990, S. 85–90.

[Giardina 84]
C. Giardina: *The Universal Imaging Algebra*, *Pattern Recognition Letters*, Bd. 2, 1984, S. 165–172.

[Goedicke 97]
M. Goedicke: *Java in der Programmierausbildung: Konzept und erste Erfahrungen*, *Informatik Spektrum*, Bd. 20, Nr. 6, 1997, S. 357–363.

[Gorlen–Orlow–Plexico 90]
K. E. Gorlen, S. Orlow, P. S. Plexico: *Data Abstraction and Object–Oriented Programming in C++*, John Wiley and Sons, Chichester, 1990.

[Haralick–Ramesh 92]
R. M. Haralick, V. Ramesh: *Image Understanding Environment*, in Arps und Pratt [Arps–Pratt 92], S. 159–167.

[Harbeck 96]
M. Harbeck: *Objektorientierte linienbasierte Segmentierung von Bildern*, Shaker Verlag, Aachen, 1996.

[Hornegger–Niemann 94]
J. Hornegger, H. Niemann: *A Bayesian Approach to Learn and Classify 3–D Objects from Intensity Images*, in *Proceedings of the* 12^{th} *International Conference on Pattern Recognition (ICPR)*, IEEE Computer Society Press, October 1994, S. 557–559.

[Hornegger 96]
J. Hornegger: *Statistische Modellierung, Klassifikation und Lokalisation von Objekten*, Shaker Verlag, Aachen, 1996.

[Jähne 97]
B. Jähne: *Digital image processing*, Springer, Berlin, 1997.

[Kawai–Okazaki–Tanaka–Tamura 92]
T. Kawai, H. Okazaki, K. Tanaka, H. Tamura: *VIEW–Station software and its graphical user interface*, in Arps und Pratt [Arps–Pratt 92], S. 311–323.

[Landy 93]
M. Landy: *HIPS-2 Software for Image Processing: Goals and Directions*, *SPIE*, Bd. 1964, 1993, S. 382–391.

[Lyon–Rao 97]
D. A. Lyon, H. V. Rao: *Java Digital Signal Processing*, M&T Books, M&T Publishing, Inc., 501 Galveston Drive, Redwood City, CA 94063, USA, Nov. 1997.

[Mundy–Binford–Boult et al. 92]
J. Mundy, T. Binford, T. Boult, A. Hanson, R. Veveridge, R. Haralick, V. Ramesh, C. Kohl, D. Lawton, D. Morgan, K. Price, T. Strat: *The Image Understanding Environments Program*, in *Image Understanding Workshop*, San Diego, CA, Jan. 1992, S. 185–214.

[Musser–Saini 96]
D. R. Musser, A. Saini: *STL tutorial and reference guide*, Addison-Wesley,

Reading, Mass., 1996.

[Niemann 90]
H. Niemann: *Pattern Analysis and Understanding*, Bd. 4 von *Springer Series in Information Sciences*, Springer, Heidelberg, 1990.

[Parker 97]
J. R. Parker: *Algorithms for image processing and computer vision*, Wiley computer publishing, New York, 1997.

[Paulus–Hornegger 97]
D. Paulus, J. Hornegger: *Pattern Recognition of Images and Speech in C++*, Advanced Studies in Computer Science, Vieweg, Braunschweig, 1997.

[Paulus–Niemann 92]
D. Paulus, H. Niemann: *Iconic–Symbolic Interfaces*, in Arps und Pratt [Arps–Pratt 92], S. 204–214.

[Piper–Rutovitz 88]
J. Piper, D. Rutovitz: *An Investigation of Object-Oriented Programming as the Basis for An Image Processing and Analysis System*, in *Proceedings of the* 9^{th} *International Conference on Pattern Recognition (ICPR)*, Bd. 2, IEEE Computer Society Press, Rome, 1988, S. 1015–1019.

[Pratt 95]
W. K. Pratt: *The PIKS Foundation C Programmers Guide*, Manning, Greenwich, 1995.

[Rasure–Young 92]
J. R. Rasure, M. Young: *Open environment for image processing and software development*, in Arps und Pratt [Arps–Pratt 92], S. 300–310.

[Standard 94]
I. Standard: *12087, Image Processing and Interchange*, International Standards Organization, 1994.

[Stroustrup 91]
B. Stroustrup: *The C++ Programming Language,* 2^{nd} *edition*, Addison-Wesley, Reading, MA, 1991.

[Tamura 83]
H. Tamura: *Design and Implementation of SPIDER - A Transportable Image Processing Software Package*, *Computer Vision, Graphics and Image Processing*, Bd. 23, 1983, S. 273–294.

[Young–Argiro–Kubica 95]
M. Young, D. Argiro, S. Kubica: *Cantata: Visual Programming Environment for the Khoros System*, *Computer Graphics*, Bd. 29, Nr. 2, 1995, S. 22–24.

[Zimmer–Bonz 96]
W. Zimmer, E. Bonz: *Objektorientierte Bildverarbeitung*, Carl Hanser Verlag,

München, 1996.