Quantitative and Qualitative Aspects of Object-Oriented Software Development

Gustav Pomberger, Wolfgang Pree

C. Doppler Laboratory for Software Engineering
Johannes Kepler University Linz, A-4040 Linz, Austria
Voice: ++43 70-2468-9431; Fax: ++43 70-2468-9430
E-mail: {pomberger,pree}@swe.uni-linz.ac.at

Abstract. Although object-oriented programming techniques have evolved into an accepted technology with recognized benefits for software development, profound investigations of qualitative and quantitative aspects about its pros and cons are missing.

This paper tries to answer crucial questions based on the experience gained by the authors and their partners in several projects where object-oriented technology was applied. These projects cover different areas like prototyping tools, information systems, real-time process control components, and development environments for object-oriented programming languages.

A case study comparing the object-oriented (C++) and module-oriented (Modula-2) implementation of a user interface prototyping tool concludes this paper.

Keywords. Design patterns, object-oriented design, object-oriented software development, application frameworks, class libraries, reusability

1 Introduction

Since the software industry can hardly efford to implement large-scale software systems twice most comparisons of conventional and object-oriented programming techniques are based on small projects. This situation served as an incentive for us to compare representative software systems.

But even reimplementations have to be considered carefully. For instance, a system should not be implemented twice by the same project team. Experience gained in the first implementation will help in the reimplementation whether or not the team starts with the conventional or object-oriented solution. Thus a system has to be implemented by two different project teams which have about the same knowledge in the beginning.

The type of a system to be built also has an undeniable influence on the comparison. Strongly algorithm-oriented systems won't benefit as much from object-orientation as systems that have to deal with various different and complex data structures.

This paper starts with a presentation of results based on several systems which have been developed in various object-oriented programming languages (C++, Smalltalk, Object Pascal, Eiffel and Oberon) and compared to conventionally

implemented systems over the past five years. The overall size of these projects is about 500,000 lines of code. The projects were carried out together with partners from industry, for example, the Union Bank of Switzerland (UBS), Siemens AG Munich, and the Austrian Industries Corporation.

After discussing qualitative and quantitative aspects derived from several case studies, we present one specific case study that demonstrates how our results were obtained.

We presuppose that the reader is familiar with the object-oriented concepts of inheritance, polymorphism and dynamic binding, as well as with principles of application frameworks (like ET++ [12] and UniDraw [10]).

2 Condensed Results

The quantitative and qualitative evaluations of various projects presented in this section were compiled by our department. The projects taken into consideration include, for example, a user interface prototyping tool (see Section 2 for details), a real-time process control system [5, 13] and software development environments such as the module-oriented SCT [1] and the object-oriented Omega [2].

In the following two subsections positive aspects of object-oriented software development are marked with a "+", negative ones with a "-". If there is no significant difference as compared to conventional software development, an "o" sign is used.

2.1 Quantitative Aspects

The numbers given for various quantitative aspects express the impact of object-orientation compared to conventional approaches. These numbers have to be seen as a percentage with reference to the conventional solution. A value of 120%, for instance, means that the quantitative measure increases by a fifth if the object-oriented paradigm is used instead of a conventional paradigm.

Impact on Source Code Size. In the case of object-oriented programs, we have to discern between the newly written source code and the reused source code:

+ Newly Written Code: 25-50% The reduction of source code that has to be written is often considered to be a very important achievement of object-oriented programming. In fact, this number directly influences the duration of a software project and its maintenance costs. Since many standard problems of a specific domain have already been solved by classes of an application framework, usage of an appropriate framework can even reduce the size of newly written source code up to 90%. Such extreme reductions are typical for small projects (about 5,000 to 15,000 lines of code) where an existing application framework has to be adjusted to specific needs. Code reductions are less in case of systems that contain many algorithm-oriented parts.

Even if there is no class library at hand reductions of the size of newly written code can be observed. These reductions are proportional to the project size. The larger the project the less code has to be written. The reason for this lies in the internal reuse of components within a project. A careful design (especially factoring out commonalities into superclasses) implies that more components can be reused as compared to conventional (e.g., module-oriented) software development.

- Overall Source Code Size: 120-300% Although less code has to be written, the overall size of the source code increases. This is caused by inherited code and indirectly

used classes. The given number may vary extremly depending on the size and cohesion of the used class library. Using classes only for data structures, for example, results in a minimal increase of the overall source code size and almost neglectible savings in code that has to be written newly. On the other hand, using an application framework implies importing numerous classes that do not contribute directly to the solution of a particular problem. But these additional classes also provide functionality for free which could only be achieved with enormous effort if no application framework is used. The generality of directly and indirectly used classes constitutes another factor that influences the size of imported code. In order to increase the generality of a class additional methods have to be provided that enlarge the code size even if they are not used.

Impact on Run-Time. More precise numbers can be given about the impact of object-oriented programming on run-time issues than can be given about the overall code size. The time elapsed during a procedure call and method call can be measured resulting in a factor. The same is true for accessing object components. Of course, garbage collection effects a program's run-time too, since searching for unreferenced objects consumes time. Implications on the overall run-time are more difficult to determine, since it depends on the quality of the underlying class library and the programming style.

- *Method Call: 105-120%* Due to dynamic binding, method calls are more expensive than procedure calls. But it has to be taken into consideration that the overhead for passing parameters and the stack-like management of procedure call chains remains the same in both cases. Thus the given numbers describe calls without parameters based on method search with indexing. So-called dynamic method searching approaches used in some pure object-oriented languages cause significantly more overhead.

- *Accessing Object Components: 100-?%* We assume that pointers to dynamically allocated memory are also used in the conventionally implemented system. The required machine code does not differ in either case (conventional and object-oriented) if the object-oriented language allows access to object components from outside. This kind of component access also occurs when object components are accessed within an object's method.

Accessing object components becomes much more expensive if method calls have to be used. In that case this operation can cost up to ten times more than a direct access. But this increase also occurs in conventional programs that are based on encapsulation and abstract data types.

- *Automatic Garbage Collection: 105-150%* Several factors determine the costs of automatic garbage collection: number and size of objects, available memory, frequency of object generation and, obviously, the method of garbage collection. Normally automatic garbage collection does not cost more than 5% of the overall run-time. This minimal overhead is made possible by sophisticated garbage collection methods that take into account that object-oriented programs generate numerous objects with short life times. Unfortunately, since many objects are generated during calculation intensive periods most garbage collection methods become active then. Thus slow downs may occur that cannot be tolerated in real-time applications. Incremental methods avoid this disadvantage. They search for unreferenced objects and dispose them quasi parallel to the actual program. This parallel garbage collection effort avoids abrupt slow downs but also retards normal execution.

o Overall Run-Time: 80-120% Though some factors cause a negative impact on a program's run-time, we could not observe a significant increase of the overall run-time. We believe that there are three reasons for this:

- Systems written in hybrid languages do not only use objects. Methods contain not only method calls but also "regular" statements. Many local variables and basic data types are used in hybrid languages. This means that no dynamic binding and access of object components occurs.
- Class libraries use highly efficient implementation techniques. For example, sets of objects are implemented by means of hash tables resulting in a constant factor required for searching an element. Usage of such classes means an increase of several magnitudes in run-time efficiency.
- Dynamic binding is less expensive than a sequence of selection statements. Conventional programs are cluttered with case statements that can be avoided in object-oriented systems.

- *Memory Requirements: 120-200%* Memory requirements of object-oriented systems depend not only on the source code size but also on method tables and dynamically allocated objects. The latter two aspects are discussed below.

The used class library exerts a strong influence on the memory requirements caused by source code. As already mentioned above the overall source code size increases due to direct and indirect reuse of classes. Finally, the compiler and linker determine the overall memory requirements. Some linkers are smart enough to identify classes and methods that are not used so that this overhead can be avoided.

o Method Tables: 101-110% The ratio of method table size and source code size depends on the number of methods provided by objects of a particular class as well as on the method size. If a class implements 20 methods, for example, all subclasses will cause at least 20 entries in the method table even though the methods are not overridden in subclasses. Thus many small methods lead to huge method tables. Nevertheless, the memory requirements of method tables are almost neglectable compared to the overall memory requirements.

- *Dynamically Allocated Objects: 100-120%* Usage of dynamically allocated objects means increased memory requirements to manage the objects. This is especially true for automatic garbage collection which requires additional information about an object's structure. If there also exists a method table that stores references to all objects, memory requirements can increase up to 20%. In case of simple memory management without garbage collection, objects are not more expensive than other dynamically allocated data.

2.2 Qualitative Aspects

This section presents qualitative aspects focused on project management and the final software product. The presented results are also corroborated by other authors, e.g., [4].

Impact on Project Management. The positive effects of object-oriented programming on several phases of the software life cycle result from the reduction of the semantic gap (developers can think in terms of real world objects), increased

reusability and thus the fact that less new code has to be written. It is no surprise that most disadvantages are related to the management of class libraries.

+ *Planning.* It can easier be determined in the planning phase of a project which subprojects/tasks will require most efforts. The reason for this lies in the reduced semantic gap and the better reusability of object-oriented building blocks. Because of the almost schematic way in which object-oriented systems are designed (especially if application frameworks are used), more time remains for planning.

+ *Organizational Effort.* Due to the reduced design and implementation efforts, smaller project teams are sufficient. Smaller teams ease the communication among the people and the coordination of subtasks.

+ *Design and Development.* Frameworks already anticipate much of a system's design. Thus elbowroom diminish which could else lead to bad design is diminished. Object-oriented programming supports a more homogeneous style so that programmers who are not involved in design decisions will be able to understand a systems's design more quickly.

Furthermore, design and implementation are more intertwined so that intermediate results will be produced within a shorter time frame. Thus project team members will experience a sense of achievement which can strongly motivate them.

+ *Division of Labor.* Object-oriented analysis and design produce class definitions which create a modularization of the system under development. The classes can be refined (i.e., implemented) almost independently by different teams or members of a team. Another kind of division of labor can be observed in larger organizations where a dedicated group is responsible for providing elementary classes and concepts.

+ *Prototyping.* Due to a reduced implementation effort, extensible prototypes (in the sense of evolutionary prototyping) for some system components can be developed within a short time. Application frameworks for graphic user interface programming also help to come up with user interface prototypes within a reasonable time if user interface prototyping tools are missing.

- *Settling-In Period.* Programmers with experience in conventional programming often have a hard time getting used to the new way of thinking which object-oriented software development requires. A couple of months are ususally required to accomplish this migration from conventional paradigms to the object-oriented paradigm.

- *Management of a Class Library.* In order to benefit from a class library it has to be kept up-to-date. The required effort directly depends on the number of users of a particular class library, its size (i.e., number of classes) and the number of projects that are based on it.

- *Project Cost Calculations.* Due to reuse of software components it becomes unclear how to split the costs among several projects. A class developed in a project causes costs which are higher if the class is designed to be reusable in other projects. But other projects benefit from this additional effort. The costs of class library management have to be treated in a similar way: they have to be shared among several projects.

Implications on the Quality of the Final Product. Object-oriented software development improves the overall quality of the produced software. Quality

improvements result from the reuse of already tested components. Subsequently, implications of object-oriented software development on crucial software quality criterions are presented.

+ *Correctness and Reliability.* Since the correctness of a software system depends on the correctness of its components, the probability of producing a correct system is the higher when more matured software components can be reused. Components of a class library usually have been reused a number of times so that their correctness can be regarded as high (ideally 100%).

Furthermore, the usage of an application framework has a positive effect on the correctness of a software system, since cookbook recipes describe of how to combine framework components in order to achieve a certain goal. Adhering to these guidelines means avoiding typical errors.

The reliability of a software system is also tightly coupled with its correctness. So reusable object-oriented components exert a positive effect on this quality aspect.

Experience has proven that the time consumed for stabilizing a software system can be reduced to a third of the time required for conventional systems.

+ *User Friendliness.* The ease of use is especially improved if a system is based on a GUI application framework. GUI application frameworks anticipate much communication between application and user. This provides functionality that would not have been implemented in conventional applications since many features that make GUIs easier to use are hard to implement. Most GUI application frameworks, for example, support undoing/redoing of commands. Furthermore, several different look and feel standards are supported by some GUI frameworks so that applications based on such a framework can easily be ported to other look and feel standards.

Offering too much functionality could also be a disadvantage. Since it is very simple to provide certain features just by reusing components, many details could detract users from what they really want to do.

+ *Maintenability.* Due to the modularity of object-oriented programs, errors can easier be detected and localized: operations associated with objects are gathered in one class. Thus testing of object-oriented programs becomes easier, too.

Since frameworks define already much of a system's design, they cause a uniformity of all systems built on top of them. Thus learning details about an object-oriented system often means less effort compared to conventional ones.

Finally, polymorphism and dynamic binding promote extensibility as they can help to avoid case statements spread over a software system.

- *Efficiency.* Object-oriented software development might have a negative influence on a program's run-time and memory requirements (see the section about quantitative aspects for details). Taking the advances in hardware into consideration, these costs can almost be neglected.

- *Portability.* Porting object-oriented systems based on class libraries that depend on a specific hardware and operating system can be considered as hard as porting such conventionally implemented systems.

Furthermore, porting an object-oriented system from one language to another is anything but trivial: a C++ program, for example, cannot be transformed into a Smalltalk program or vice versa simply by syntactical changes. The different ways of thinking in both language worlds have to be matched.

3 A Representative Case Study

We pick out the case study where a conventionally implemented user interface prototyping tool is compared to its object-oriented solution. This case study is representative since the two software systems have the "critical mass" as far as complexity is concerned. Furthermore, the software systems have been implemented by different teams—one at the University of Zurich and the other at the University of Linz. The case study mirrors the results presented in the previous section.

The conventionally implemented User Interface Construction Tool (UICT [1], implemented in Modula-2) and the object-oriented Dynamic Interface Construction Environment (DICE [6, 7], implemented in C++ with the application framework ET++ [3, 11, 12]) form the basis of one such large-scale case study used to evaluate the promises of object-oriented programming as well as its pros and cons compared to module-oriented software system development. (Both projects have been supported by Siemens AG Munich.) UICT and DICE serve for the prototyping-oriented development of graphic user interfaces under UNIX and the X11 window system.

The comparison of UICT's and DICE's development is based on concepts which the tools have in common. Table 1 shows these concepts and their realization in UICT and DICE.

Concepts	Realization in UICT	Realization in DICE
Prototype Specification	User Interface Description Language (Text-Oriented)	Graphic-Oriented Specification Formalism
Prototype Representation (Data Structures)	Intermediate Language Based on Arrays	(Sub)classes of an Application Framework
Prototype Simulation	Interpreter	Simulation Framework
Code Generation	Conventional Generator	Code Generation Framework

Table 1 Common UICT and DICE concepts together with their realization

We pick out the prototype representation component as typical example that corroborates the results presented above.

Prototype Representation—Employed Data Structures. The qualitative and quantitative comparison of UICT's and DICE's internal prototype representation considers the following aspects:

- We compare employed data structures as well as their management and investigate the influence of different ways of thinking (due to the module-oriented and object-oriented paradigm) on the internal prototype representation.
- The internal prototype representation plays an important role in each user interface prototyping tool. Effects of extensions/modifications of the internal representation are contrasted.

- The particular software components that implement the internal prototype representation are compared (using lines of code as a metric for a quantitative comparison). This comparison particularly considers the degree of software reuse.

Data Structures and Their Management. UICT and DICE have to store a prototype specification in an appropriate format. This is the precondition of a later simulation and code generation. We induce the following general statements from the comparison of employed data structures:

Data structures employed in the module-oriented implementation are not *active*. This means that modules dedicated to internal prototype representation store only *descriptions* of user interface objects, but not objects in an object-oriented sense. This causes other modules to implement routines that are based on these descriptions—descriptions (= data) and operations (= routines that manipulate and interpret these descriptions) are *separated*. This decreases the extensibility of the data structures.

A typical object-oriented implementation unifies data and operations and makes data structures active. This means that the description of an object's behavior forms a unit (the attributes of an object that constitute its state and all methods that determine its functionality). Furthermore, common behavior is factored out into an abstract class so that other components can be based on that class. Therefore modifications of common behavior are easier to accomplish.

Implication: The employed data structures and module/class architecture primarily result from different ways of thinking in the design of module-oriented and object-oriented software systems.

These general statements are drawn from UICT's and DICE's internal prototype representation:

Data Structures and Their Management in UICT: We first try to explain the main reason for UICT's overall module structure in order to outline the influence of such a modularization on the internal prototype representation: If a system is modularized, only the system's functionality is taken into account in most cases. UICT is a typical example. It consists of modules that *parse* a prototype specification, modules that *handle* tables (=the internal prototype representation), modules that *simulate* a prototype, and modules that *generate* code. Modules that handle an intermediate language stored in arrays (called UI Tables) realize UICT's internal prototype representation. All other modules are built around the table handler modules. (E.g., prototype specifications in UISL are parsed by Translator modules and then transformed and stored in UI Tables.)

A UI Table logically consists of two parts: an *object list* and a *symbol list*. The object list is a set of descriptions of user interface building blocks that are specified for a particular prototype. Thus an object list contains descriptions of user interface elements supported by UICT (e.g., menus, buttons, windows). The symbol list stores all names used in the prototype specification and for each name a reference to its description in the object list. Two modules, called *Table Handler* and *Symbol Table Handler*, manage UICT's prototype representation. This fine-grained modularization is based on the two logical parts of a UI Table.

Data Structures and Their Management in DICE: One design goal of DICE's internal prototype representation was that classes representing user interface elements should not only consist of their attributes described in instance variables but also of methods that determine their functionality, so that data and operations are really unified.

Another important design goal of DICE's internal prototype representation was to define an abstract class that factors out common behavior of user interface elements. Thus classes implementing graphic editor components as well as classes which initiate attribute definition (see below) of a user interface element, simulation of the particular prototype, and code generation can be based on such an abstract class. This is a precondition of a satisfying extensibility of the overall system.

We designed an abstract class called DICEItem for the reasons stated above. Subclasses of this abstract class represent specific user interface elements. Data management (i.e., management of several DICEItem instances) is accomplished by means of already existing ET++ classes for managing object collections.

Extensions of the Prototype Representation. By "extensions of the prototype representation" we mean adding new interface elements that are to be represented (this is the most important extension in connection with user interface prototyping tools). In this section we discuss the impact of such extensions on other components of the particular user interface prototyping tool.

In UICT's implementation object descriptions can be accessed from any other module in the system that imports the proper routines and data types from the modules that handle these descriptions. Changes (i.e., new kinds of object descriptions) to modules that handle these descriptions affect the whole software system. (Only changes to data structures that are internally employed by the table handler modules in order to store object descriptions do not affect the overall system.)

In DICE's implementation abstract classes define the common behavior of objects realizing the prototype representation. Other components of a system can be based on these abstract classes. Subclasses of the abstract classes describe specific object behavior. The software system is so flexible because new kinds of user interface elements can be added as subclasses to abstract classes without affecting the other components of the system.

Implications: The concepts inheritance, polymorphism and dynamic binding as offered by object-oriented programming languages are the preconditions for building extensible and reusable software systems: Inheritance opens the possibility of incrementally extending and adapting object descriptions of abstract classes that factor out common behavior. This is done in subclasses without changing the code of the abstract class. Polymorphism and dynamic binding allow the implementation of software components that are based on abstract classes. Thus these components are independent of specific object types.

Extensibility Aspects of DICE's Internal Prototype Representation: We illustrate the extensibility of DICE's components that are based on DICE's internal prototype representation (class DICEItem). We choose the attribute definition of user interface elements, which is part of DICE's specification component, as our example.

Attributes that describe a user interface element's behavior are defined in appropriate dialog boxes. Let us sketch the attribute definition of user interface elements from the user's point of view as a precondition of understanding its object-oriented implementation described below: In order to set attributes of user interface elements, one selects the particular user interface element and chooses the menu item "Item Attributes..." from a DICE-specific menu. A dialog box is opened where element-specific attributes are manipulated.

The implementation of attribute definition is based on the abstract class DICEItem: The crucial point is that classes comprising DICE's specification component should only implement a *framework* for attribute definition, i.e., handle

mouse events, menu selections and the opening/closing of a dialog box in which attributes are manipulated. Only the window contents (i.e., the specific attributes) of the dialog box must be provided by the particular user interface element. The update of specific attributes which may be changed in the dialog box has to be accomplished by the particular user interface element, too, since attributes are instance variables of each user interface element. So the abstract class DICEItem defines two (abstract) methods for attribute definition called GetAttributesDialog and UpdateAttributes.

Arbitrary new interface elements can be added as subclasses of DICEItem. They just have to override the element-specific methods for attribute definition GetAttributesDialog and UpdateAttributes. Furthermore, instance variables representing their attributes must be added. Algorithms for attribute definition as implemented in the specification component still work since they are based on the abstract class DICEItem.

Thus typical object-oriented programming techniques (inheritance, dynamic binding, and polymorphism) are the precondition for the implementation of a framework for attribute definition.

Methods of DICEItem for simulation and code generation are designed in an analogous way. So simulation and code generation frameworks can be implemented based on DICEItem. They need not be changed if new user interface elements are added as subclasses of DICEItem.

Software Reuse. The comparison between UICT and DICE components that realize the particular internal prototype representation points out significant differences between module-oriented and object-oriented systems regarding reuse of already existing software components (see the discussion of the "Impact on Source Code Size" in Section 2.1).

We use the written lines of code and the average number of lines of code per routine/method to directly compare UICT's and DICE's components (see Table 2).

The amount of code that has to be written in UICT's Table Handler Module is about seven times as much as in DICE's component for internal prototype representation. The complexity of routines can be cut back to about one third in the object-oriented implementation. The main reason for such code and complexity reductions is the high reusability of application framework components.

The module-oriented user interface prototyping tool does not even reuse existing modules for managing data structures like lists, etc. Everything is implemented from scratch. This is done in such a way that modules implemented for that purpose in UICT are again very specific.

The object-oriented realization of the internal prototype representation reuses as much code as possible from the application framework it is based on. In case of DICE's prototype representation component, we can also calculate the ratio of reused and newly written code (see Table 3).

This ratio (newly written lines of code : reused lines of code = 1052 : 1307 = 1 : 1,24) strongly depends on the application framework that is used and the originality of the problem at hand.

	UICT's Prototype Representation Component	DICE's Prototype Representation Component
written lines of code	7600	1052
routines/methods	159	62
lines per routine/method	47,8	17,0

Table 2 Comparison of UICT's and DICE's prototype representation components based on common metrics

	DICE's Prototype Representation Component
newly written lines of code	1052
reused from ET++ classes	1307

Table 3 Ratio of reused and newly written code

4 Conclusion

The comparison of module-oriented and object-oriented system development demonstrates that object-oriented programming techniques are important techniques to produce software components that are open for extensions and thus reusable. Thus software quality can be raised and the amount of code to be written can be reduced.

However, the object-oriented programming paradigm is not sufficient for the achievement of systems that are extensible/reusable without problems (see, for instance, [8, 9]). Extensibility/reusability still has limits that cannot satisfy a system developer.

5 References

1. Bischofberger W., Pomberger G.: Prototyping-Oriented Software Development—Concepts and Tools; Springer Verlag, 1992.
2. Blaschek G.: Object-Oriented Programming with Prototypes; Springer Verlag, 1994.
3. Gamma E., Helm R., Johnson R., and Vlissides J.: Design Patterns—Microarchitecturs for Reusable Object-Oriented Software; Addison-Wesley, 1994.
4. Loves T.: Object Lessons; SIGS Publications, 1993.
5. Plösch R., Weinreich R.: An Extensible Communication Class Library for Hybrid Distributed Systems; Proceedings of TOOLS Pacific '92 conference, Sydney, 1992.
6. Pomberger G., Bischofberger W., Kolb D., Pree W., Schlemm H.: Prototyping-Oriented Software Development, Concepts and Tools; in Structured Programming Vol.12, No.1, Springer 1991.

7. Pree W.: Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools; doctoral thesis, University of Linz, 1992.
8. Pree W.: Reusability Problems of Object-Oriented Software Building Blocks; EastEurOOPe'91, Bratislava, Czecho-Slovakia, September 15-19, 1991.
9. Taenzer D., Ganti M., Podar S.: Problems in Object-Oriented Software Reuse, Proceedings of the 1989 ECOOP, July 1989.
10. Vlissides J.M.: Generalized Graphical Object Editing; PhD Thesis, Stanford University, 1990.
11. Weinand A., Gamma E., Marty R.: ET++ - An Object-Oriented Application Framework in C++; OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.
12. Weinand A., Gamma E.: The GUI Application Framework ET++; in Object-Oriented Software Frameworks (ed. Ted Lewis), Prentice Hall, 1994.
13. Weinreich R.: Concepts and Techniques for Object-Oriented Software Development—Illustrated by an Application Framework for Process Automation; doctoral thesis, University of Linz, 1993.