

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000

Figure 1 Computer technologies and relative performance per unit cost

In order to produce a hardware product, a hardware engineer picks up these building blocks and assembles them together. This way increases the productivity and reliability of the product since each chip has been well tested and fabricated massively.

However, there was no dramatic change in software production. Each new software product required software designers and programmers start from scratch and produce program code line by line until the program was finished. One critical progress of software development for the last 50 years is the level of programming languages being used. High-level programming languages offer several important benefits improving programming productivity thanks to compiling techniques. Nevertheless, the line by line fashion of software production is still dominating the current practice and becoming the bottle-neck of rapid developing quality software with low cost.

This year is the 32nd anniversary of software engineering. The first NATO software engineering conference was held at Garmisch, Germany, on October 7-11, 1968. According to ANSI/IEEE standard 610.12-1990, software engineering was defined as the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software. After 30 years of practice, we have a much better understanding of the activities involved in software development. We have developed methods of software specification, verification, design and implementation. Nevertheless, many large software projects are still over-budget and behind the schedule. New technologies in computer hardware and new application areas place new demands on software engineers. People realized that it was not appropriate to use "software crises" to describe the problems facing software community, since the word "crises" is defined as "a turning point in the course of anything; decisive or crucial time, stage or event." Yet there was no "turning point" in the last 30 years. [Pressman 97] describes the phenomena as software affliction, a long-lasting pain or distress. Software engineering still has a long way to go.

Can we build software in the way of classic engineering disciplines? Can we develop software much faster than we do now with lower cost? Why is it important to build software system based on pre-built, reusable components? What is a component? How does a software designer build a reusable component? How can we ensure that the right components can be found and used? How can we guarantee that correct product can always be obtained by assembling correct components? This paper addresses these questions after looking into the 40-year computing practice and criticizing the 30-year software engineering practice. The proposed solution to these questions lies in the concept of highly reusable software components.

Nineteen research and development goals are proposed which fit well into an upper-level software engineering course as student projects at small colleges.

ESSENCE OF ENGINEERING

Engineering refers to the profession devoted to designing, constructing, and operating the structures, machines, and other devices of industry and everyday life [Encyc 98]. For instance, chemical engineering is dealing with the design, construction, and operation of plants and machinery for making such products as acids, dyes, drugs, plastics, and synthetic rubber; aerospace engineering comprises the design and production of aircraft, spacecraft, and missiles. Software engineering, similarly, is the field of computer science that deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers [Ghezzi 91].

One of the essential characteristics of engineering disciplines is to build a product by assembling pre-made, standard components. Before the industrial revolution (1750-1850), there was no factory system of large-scale machine production. Products were typically made by handcraft through custom-building fashion. That is, one craftsman ? a mechanic, a carpenter, a watchmaker or a shoemaker, etc. ? built every part of the product by himself according to the user requirements. Dramatic changes in the social and economic structure took place as inventions and new technology created modern industries after the middle of 19 century. Nowadays custom-building fashion disappears in almost every engineering discipline. A variety of standard, pre-manufactured components are available for reuse when a new product is designed. A new component-based product has higher quality and lower cost than custom-built product since standard reusable components have been produced massively and are well tested for a long time in different applications.

It is the common practice to use pre-fabricated components in designing and implementing new engineering products. In designing a new car, the engineers will more likely to adopt existing radio, tires, lights and other standard parts rather than build these components from scratch. Civil engineers build a new house by reusing standard windows, furnace, and air-conditioning systems. Computer hardware engineers have a vast collection of integrated circuits (IC) or chips reusable for new product. In order to build a seven-segment display, for instance, hardware engineers look into the library of reusable hardware components, just pick up chip 7448 as a decoder and connect it to digit seven-segment displays. It is a usual practice for a college student to assemble a personal computer based on different hardware components: processor, motherboard, RAM chips, disk drives, modem, keyboard, mouse, etc.

Modern industries depend highly on large-scale manufacturing and engineering process as opposed to hand crafts, the designing and hand fabrication of functional objects. Any engineering discipline requires three pre-conditions: (1) the fundamental theory exist as a guidance of the engineering practice; (2) widely accepted standards exists; (3) highly reusable components available so that massively production of high quality products becomes possible. If we apply these criteria to software engineering, we find great challenges still remain especially

regarding highly reusable software components. Our ability to build new software cannot keep pace with hardware advances and application demand.

SOFTWARE ENGINEERING: AT ITS 32ND ANNIVERSARY

The growth of software engineering started from programming. In the early days of computing, the problem of programming was viewed essentially as how to place a sequence of instructions together to get the computer to do something useful. The programmer wrote code line by line instructing the computer in every single action from loading data into accumulator, increment the program counter, doing the operation, and so on to storing result into a particular memory cell. Such programming tasks were difficult because these low-level instructions, machine code or assembly code, were so tedious and error prone. Higher-level programming languages were invented in the late 1950s to make it easier to program. Since then, thousands of programming languages have been designed and implemented. However, the activity of programming today is essentially the same as it was: developing a program line by line until it is finished.

Structured Programming

Structured programming has a narrow definition and a broad definition. By the narrow definition, structured programming means programming without using goto statements, programming using only while loops and if statements as control constructs and designing using a top-down approach. The broad definition of structured programming refers to any software development technique that includes structured design and results in the development of a structured program. Structured programming allows programs to be broken down into blocks or procedures which can be written without detailed knowledge of the inner workings of other blocks, thus allowing a top-down design approach or stepwise refinement. Stepwise refinement is an iterative process. At each step, the problem to be solved is decomposed into sub-problems that are solved separately. The sub-solutions that comprise the solution of the original problem are then linked together by means of simple control structures.

The adoption of structured programming was an important milestone in the development of software engineering because it was the first step away from an undisciplined approach to software development. However, apart from goto statements, there are also several other constructs in programming languages which are inherently error-prone, for instance, floating-point numbers, pointers, dynamic memory allocation, parallelism, recursion, interrupts, and so on. Stepwise refinement, on the other hand, cannot provide a systematic strategy for finding a solution, not to speak of a good solution. Sometimes, the top-level function may not exist. Structured programming is an effective technique for small-sized programs. But it fails to scale up to large system development. It does not help designers reuse components from previous applications or design reusable components when applied to larger programs. It does not support information hiding. There is a premature commitment to the control structures that govern the flow of control among modules.

Computer-Aided Software Engineering

Since the early 1980s, hundreds of computer-aided software engineering (CASE) tools have been developed with the hope of dramatically increasing software productivity, as various computer-aided design (CAD) and computer-aided manufacturing (CAM) have proved to be successful in other engineering disciplines. Figure 2 lists a number of different types of CASE tools with specific examples of each tool.

Tool type	Examples
Management tools	PERT tools, estimation tools
Editing tools	Text editors, diagram editors, word processors
Configuration management tools	Version management systems, change management systems
Prototyping tools	Very high-level languages, user interface generators
Method support tools	Design editors, data dictionaries, code generators
Language processing tools	Compilers, interpreters
Program analysis tools	Cross-reference generators, static analyzers, dynamic analyzers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Document-generating programs, diagram editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

Figure 2 Classification of CASE tools

However, the current CASE products have not led to the high level of productivity improvements that were predicted by their vendors. As shown in Figure 3.1, tools do exist for specific stages of software life cycle; but they are not integrated together, nor have the benefits of cross-platform support. Furthermore, CASE does not change the current practice of software development essentially: writing code line by line until it finishes. Software engineering has not yet benefited from automation to the same extent as other engineering disciplines.

Object-Oriented Programming

Object-oriented programming has been recognized as a new programming paradigm in contrast to the alternative programming paradigms, which are predominantly procedural, logical,

and functional. Object-oriented programming is based upon encapsulation, inheritance, and polymorphism. Encapsulation packages data and the operations that manipulate the data into a single named object. Inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated from them. Polymorphism enables a number of different operations to have the same name, reducing the number of lines of code required to implement a system. It is reported that during the first half of the 1990s, object-oriented software engineering became the paradigm of choice for many software product builders and a growing number of information systems and engineering professionals. As time passes, object technologies are replacing classical software development approaches [Pressman 97].

Object-oriented programming stresses information hiding, reusability, and new analysis and design methodologies. However, object orientation is more oriented towards analysis and design rather than coding. Object orientation is a way of thinking, not tied to particular language or coding stage. It focuses on objects as encapsulations of state plus behavior with clear interface. Inheritance in object orientation provides reusability, extendibility, and lower maintenance cost. But the fundamental activity of programmers is still unchanged: writing code line by line. Furthermore, object orientation is difficult to learn and apply in practice.

In summary, software engineering has made great progress during the last 30 years with many innovative technologies invented and applied including structured programming, CASE technology, and object-oriented technology. However, the essential style of developing software remains the same: the programmers write code line by line in the coding stage. Such a pre-industry style of production is the key factor for the un-match between hardware productivity and software productivity. The revolutionized changes in software development, like the industrial revolution to classical engineering disciplines, have to be introduced:

- The handcraft fashion of software development should be replaced by engineering methods.
- The line-by-line-coding style should be replaced by component-based development.
- The syntax-based programming should be replaced by assembling components based on their syntax and semantics.

WHAT IS A COMPONENT?

Component-based software engineering (CBSE) intends to build large software systems by integrating pre-built software components. The high productivity is achieved by using standard components. The principles of CBSE can be best described by the following two guiding principles: reuse but do not reinvent (the wheel); assemble pre-built components rather than coding line by line. Obviously we must answer first: what is a component?

[G 1] Give an appropriate, exact, and concise definition for software components in CBSE.

A number of characteristics of a component should be captured by the definition. A component should be a piece of self-contained code. A consequent question will be: what kind

of code? Is it a piece of compilable source code? Executable code? Assembly code or byte-code? A related research topic is expressed in the following research goal:

[G 2] Investigate the granularity of components.

Components are basic building blocks in component-based software engineering. The granularity of these building blocks affects component reusability, productivity and maintenance. In traditional programming paradigm, the basic building blocks are data structures like integers, arrays, pointers, etc. and library functions. In object-oriented programming, the basic building blocks are objects, which are implemented as classes in C++ and Java. The reuse of individual classes will not bring significant productivity leaps because the granularity is small. Java provides package-level reusability so that a collection of classes can be reused by inheritance. The reusable components in CBSE should have larger granularity than traditional objects and packages.

[G 3] Find out different component structures.

Based upon data like Boolean values and integers, we can build data structures such as lists, trees, or queues solving common problems and abstracting from the building blocks involved. Design patterns can be viewed as "object structures" that capture common object-oriented solutions and abstract from the underlying building blocks, namely the objects. Similarly we should identify and investigate component structures based upon reusable software components.

[G 4] Compare hardware components and software components.

Logically computer hardware and software are equivalent in terms of their computability. That is, any operation performed by software can also be built directly into the hardware. On the other hand, any instruction executed by the hardware can also be simulated in software. Typical hardware components are integrated circuits (ICs) including medium-scale integration (MSI), large-scale integration (LSI) and very large-scale integration (VLSI) circuits. These chip-level components are possible since logic circuits have well-defined input-output functions which do not require interaction with outside world. The communications among hardware components are supported by system bus. A software component, on the other hand, is usually difficult to be expressed as simply a function from input domain to output co-domain. Moreover, some software components must interact with users, thus they cannot be treated as simply black boxes like their hardware counterparts.

[G 5] Investigate the common features and differences between object-oriented technology and CBSE.

Traditional programming uses individual instructions as building blocks. Object-oriented programming allows objects as components. Is CBSE replacing object-oriented technology? Is object orientation a necessary precondition for CBSE? In an object-oriented programming language, the only method to compose existing components into a new component is through inheritance. In Java, for instance, we make use of existing API classes and interfaces by extends and implements clauses:

```
public class My_Class extends Existing_Class implements Existing_Interface {...}
```

The essence of such clauses is programming by extension. Component-based software engineering needs more powerful composing methods than inheritance and extension. Object orientation is not a necessary precondition of CBSE. On the other hand, both object orientation and CBSE emphasize software reuse.

COMPONENT REPRESENTATION

[G 6] How can a component be documented so that it is easy to use?

The documentation of a component has to be different from the documentation of traditional software because it must not only describe what the component does but also what are the interactions between the components and its users. Thus the description should contain two parts: (1) static functionalities; (2) dynamic behaviors, i.e., the communications with other components. It is not enough to provide only the source code of components. Formal notations plus hypertext, visualization, simulation and prototyping will be helpful.

[G 7] Establish the formal computation model of components.

We need a more rigorous basis for describing software components. At the very least, we should be able to define precisely the extension and intension of software components. [Wegner 97] proposes a new computation model: interactive machine, which was proved to be more powerful than traditional Turing machines. We can define a component as a 6-tuple interactive machine: $C = (Q, I, f, O, q_0, F)$ where Q is the set of states, Q can be infinite; I is the set of input streams; f is the transition function; O is the set of output streams; q_0 is the initial state; F is the set of final states. The composition, transformation, minimization and grammar correspondence of such component machines can be discussed following the traditional formal language theory. Interactive machines provide a potential formal semantics for component algebra.

[G 8] Investigate the metrics of components.

Software metrics provide measurement to assess the quality of software products. We need to investigate the appropriate metrics for component based software development. The primary objectives for component metrics include the following:

1. to better understand the quality of components;
2. to assess the effectiveness of component assembling;
3. to improve the quality of components and final products.

To be useful in component-based development, component metrics must be simple and computable, consistent and objective. They capture the characteristics of a component including easy to implement, amenable to test and simple to modify. High-level design metrics should be combined with function-based and event-based metrics at component level.

[G 9] Find out appropriate representation scheme for components by examining different representation methods.

Over the past years, several knowledge representation schemes have been proposed and implemented in artificial intelligence field. These representation methods can be used in representing software components.

1. Logical representation schemes. Logic formulas are used to represent components and inference rules and proof procedures are used to express the composition of components.
2. Procedural representation schemes. Procedural schemes represent components as a set of instructions for solving a problem. This contrasts with the declarative representations provided by logic representation schemes.
3. Network representation schemes. Network representations capture components as a graph in which the nodes represent components and the arcs represent relations or associations between them. Unified Modeling Language (UML) might be a good starting point to investigate the proper representation of components based on graphical notations.
4. Structured representation schemes. Structured representation languages extend networks by allowing each node to be a complex data structure consisting symbolic data, pointers to other frames, or even procedures for performing a particular task.

Each of these representation schemes has its own strengths and weaknesses. We need to tailor these representation methods or combine them for the purpose of best representing software components.

A COMPONENT SPECIFICATION LANGUAGE

[G 10] Investigate the feasibility and pros and cons of adopting UML as a component description language.

The Unified Modeling Language (UML) has generated interests in component modeling and object-oriented development. UML is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. But UML lacks of formal semantics. It only defines the notation for various diagrams; it does not provide adequate process models, data models, and interface-flow diagrams to properly model real-time, interactive and hybrid systems. Further investigation is needed on evaluating UML's capability as a component description language.

[G 11] Investigate the possibility of adopting IDL as a component specification language.

Interface Description Language (IDL) is a C++ like language specifying the services supplied by a certain component. IDL describes the syntactic part of a component's interface, but it does not help to specify semantics. Further work is needed to investigate IDL's compositional mechanisms for gluing components together.

[G 12] Design a formal specification language for reusable software components.

Investigate the necessity of introducing a meta-language as a specification language for components, which is different from the programming language implementing the components. This specification language is the basis for all inferences that will be made by the system and therefore determines what can be ultimately expressed. A number of questions must be

answered in designing such a specification language, to name but a few: What are the most fundamental operations among components? How to formalize component communications? What semantic basis should be set up for such a specification language?

A COMPONENT COMPOSING LOGIC (ALGEBRA)

In formalizing component-based software development, a formal system needs to be established for specifying and verifying component systems. There are two methods leading to the formal system for specification and verification: (1) Logical method: specification is described as formulas in certain formal logic system; verification is done by theorem-proving or model checking based on the denotational semantics of the formulas. (2) Algebraic method: specification is an algebra generated by system objects and operations; verification is conducted by comparing system behavior based on the operational semantics of the specification. According to Lambek-Lawvere correspondence, logical method and algebraic method are essentially the same in terms of isomorphism.

[G 13] Establish a formal logic calculus for component-based software development.

Such a formal logic system should satisfy the following three properties:

- 1 *Expressiveness*: It should be expressive enough with higher-level of abstraction so that it gives designers more freedom at specification and design stages while postpones the decisions of implementation details.
- 2 *Composability*: It allows a large system to be decomposed into small subsystems so that the correctness of the system is transformed into the correctness of subsystems and the correct composition.
- 3 *Decidability*: The logical system should be decidable so that machine-aided verification or automatic verification tools can be developed.

[G14] Investigate the algebraic structure of components.

If we interpret components as objects and compositions as morphisms, all components could form an algebraic category. Thus theory and results in category theory could be used to investigate components and their properties. According to Lambek-Scott correspondence, any deduction system can be treated as an algebraic category if we adopt propositions as objects and inferences as morphisms. If we interpret certain composing operation as reduction, an interesting consequent question is whether this reduction satisfies Church-Rosser property.

A SOFTWARE PROTOTYPE FOR COMPONENT FRAMEWORK

A component framework is defined as a triple: $CF = (CC, CAT, SAT)$ where CC is a component (logic) calculus; CAT is the algebraic category defined by CC ; $SAT \subseteq CC \times CAT$ is a satisfaction relation. The initial work has demonstrated strong evidence that all four logical axioms and one non-logical axiom scheme of generalized concurrent calculus [Wang 92] are

valid for reasonable software components and compositions. The key issue to be investigated is to decide the structured axiom set of the component calculus.

[G 15] Develop a software prototype for component framework.

A software prototype is usually constructed for early assessment of a software product. For component framework, the prototype should provide a visual design environment for software engineers to manipulate software components. It has two major functions: (1) to develop components for reuse; (2) to develop software products based on existing components. Figure 2 presents a tentative graphical user interface for component framework. The first window contains common operations on components. For instance, the submenu Project includes menu-items like New Project, Open, Close, Save, Save As, Versions, Print etc. The submenu Component includes menu-items New Component, Open, Close, Operations and so on. The submenu Operations again contains all component manipulating mechanisms defined in the component algebra.

The *Component Library* window can be used as a component base. All existed components are stored in this component base. Different searching methods are provided for users to pick up desired components for manipulation.

[G 16] How to develop a component with extensibility and reusability?

[G 17] What is the systematic way to classify and store existing software components in the component library?

[G 18] How to search and retrieve software components effectively?

[G 19] How to simulate software components?

DISCUSSION

There are two basic activities in component-based software development: First, develop components for reuse. The production process model for this activity involves component specification, design, coding, testing, and maintenance. Second, develop software with components. The component search engine is used in this activity to obtain the appropriate components while the composing logic is applied to prove the correctness of component integration. Due to the similar reasons argued in [Wegner 97], such a component composing logic might not be complete, but it should be sound. Moreover, only syntax proofs will not enough. We need semantic information in verification and we need simulation in validation process.

The author has proposed 19 research topics and goals to investigate fundamental issues in component-based software engineering. They can be incorporated in an upper-level software engineering course for undergraduate students majored in computer science or for graduate students specialized in software engineering in small colleges. The ultimate goal of such research is to provide a scientific and engineering basis for designing, building, and analyzing reusable software components and their systematic composition methods. In addition to the research

agenda discussed in previous sections, there are certainly more research topics that are worth further exploring. Your comments and suggestions are highly appreciated.

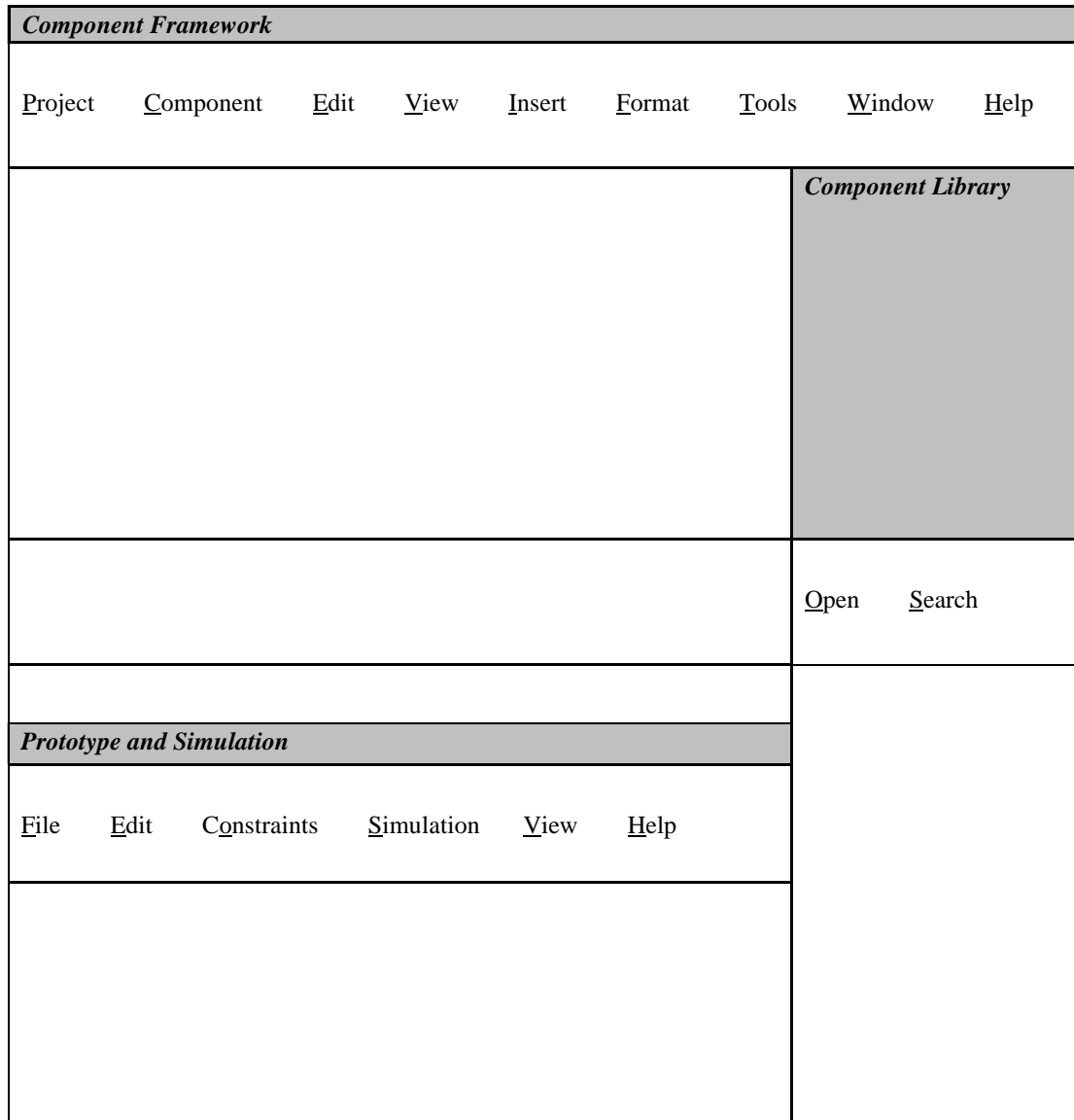


Figure 2: A GUI for component framework

REFERENCES

[Ambler 97] Ambler S. W., What's Missing from the UML? Object Magazine, October 1997.

[Encyc 98] Infonautics Corporation, On-line Encyclopedia, <http://www.encyclopedia.com>, 1998.

- [Luger 98] Luger G. F. and Stubblefield W. A., *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*, Addison Wesley Longman, Inc. 3rd ed., 1998.
- [Ghezzi 91] Ghezzi C., Jazayeri M. and Mandrioli D, *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [Hand 92] Henderson-Sellers B., *A Book of Object-Oriented Knowledge*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Leav 97] Leavens G.T., and Sitaraman M., *Proceedings of Foundations of Component-based Systems Workshop*, Zurich Switzerland, September 1997.
- [Moss 96] Hanspeter Mossenbock, *Trends in Object-Oriented Programming*, ACM Computing Surveys 28A(4), December 1996.
- [OMG 98] Object Management Group, *CORBA 2.2/IIOP Specification*, OMG Formal Documentation, <http://www.omg.org/library/specindx.htm>.
- [Pat&Hen 98] Patterson D.A. and Hennessy J.L., *Computer Organization & Design: The Hardware/Software Interface*, 2nd Ed., Morgan Kaufmann Publishers, Inc., 1998.
- [Pressman 97] Pressman R.S., *Software Engineering, A Practitioner's Approach*, 4th eds., McGraw-Hill, 1997.
- [Somme 96] Sommerville, I., *Software Engineering*, Addison-Wesley, 1996.
- [UML 98] Rational Software Corporation, *Unified Modeling Language: UML Resource Center*, <http://www.rational.com/uml/index.shtml>.
- [Wang 92] Wang J. A., *Concurrent Calculus and Its Applications*, Ph.D. thesis, 1992.
- [Wang 98] Wang J.A., *Object Orientation in Computer Graphics*, seminar presentation, April 1998.
- [Wegner 97] Wegner P., *Why Interaction is More Powerful than Algorithms*, CACM, May 1997.