

# Reusable Object-Oriented Design

G. Butler            L. Li            I.A. Tjandra

Centre Interuniversitaire en Calcul Mathématique Algébrique  
Department of Computer Science  
Concordia University  
Montreal Quebec H3G 1M8 Canada  
{gregb,li,ono}@cs.concordia.ca

## Abstract

Software design is a difficult creative task learnt from long experience. Reusable object-oriented design aims to describe and classify designs and design fragments so that designers may learn from other peoples' experience. Thus, it provides leverage for the design process. This paper surveys the field, discussing software architectures, application frameworks, design patterns, and the design of class libraries. The field is young with many open problems that still need to be researched.

## 1 Introduction

The drive for productivity in the software industry is forcing major changes in the ways that software development and maintenance are being done. Traditionally, over 80% of total expenditure is directed to *software maintenance* (including corrective, adaptive, and perfective maintenance), and about 60-70% of the effort of maintenance is directed towards *understanding* the software (requirements, architecture, design, and code). Hence, lowering the effort and costs of maintenance and understanding are the primary means to increased productivity. Software reuse [9, 27, 67, 64] amortizes the cost of developing a component across many projects and supports better quality of products through more effort in inspections and testing. Reuse applies not only to source-code fragments, but to all the intermediate products generated during software development, including requirements, documents, system specifications, and designs. Systematic reuse of existing software components to construct new systems has been successfully practiced by organizations since at least 1979 [50, 51].

Software design is a difficult creative task learnt from long experience [18]. Reusable object-oriented design aims to describe and classify designs and design fragments so that designers may learn from other peoples' experience. Thus, it provides leverage for the design process. This paper surveys the field, discussing software architectures, application frameworks, design patterns, and the design of class libraries. The field is young with many open problems that still need to be researched.

Design is the activity to produce a description of *how* to perform a task which meets the customers' requirements and any other constraints imposed by the context in which the task is to be performed. That is, software design produces a mechanism to perform the task, and that mechanism is later realised in a programming language. There are many methodologies for designing software [13, 18, 26, 39, 40, 44, 66, 76, 80, 81]. They chiefly address designing a system from scratch rather than design with reuse. Design is an example of a "wicked problem" [18, pp.19-21]. In particular, the requirements for a system may not be fully understood until the design is complete; there is no right answer to the problem, just a vague distinction between good and bad designs; and there is no way to converge to a good design, one postulates a tentative design and by analysing its merits and deficiencies may refine it to a complete working design.

Parnas' writings [60, 61, 62] have led to the consensus that “design for change” is an overarching principle for software design [32, p.65], and that information hiding is an essential step to achieving design for change. The object-oriented paradigm stresses information hiding through the use of private state variables for objects and loose coupling through message passing [45], so it is well-suited to design for change. Meyer [55] makes the argument that object-oriented design is better for reuse than functional design. He uses small examples to illustrate the argument that by focussing on data rather than process there are more reusable components: the objects are common across applications whereas functionality is not commonly reused.

The reusable design artifacts are

- software architectures, which describe the structure of systems at the level of gross organization and global control;
- application frameworks, which are actual implementations of those architectural components common to a family of applications, including the design and implementation of global control and the global division of responsibilities;
- design patterns, which are micro-architectures or mechanisms that resolve one issue in design; and
- class libraries, which are collections of classes and incorporate the design of interfaces and class hierarchies.

These artifacts may be classified as *concrete* if they provide an actual realisation of the mechanism as well as the design of the mechanism; otherwise they are *abstract*. Application frameworks and class libraries are concrete; software architectures are abstract; design patterns are abstract, but they are often accompanied by examples which realise the design pattern in a specific application.

We should be careful to distinguish *design patterns*, which are micro-architectures, from *patterns* [41], which document how to reuse or apply a design. A discussion of the different uses of the word “pattern” is provided in [75].

Every designer can benefit from a knowledge of software architectures and design patterns, and from familiarity with class libraries for domain-independent components such as data structures and user interfaces. However, more systematic reuse of domain-dependent design artifacts brings more benefits. Systematic reuse [33] requires a change in the whole software process, from requirements analysis to maintenance, and impacts the whole organization and the way it does business. Costing, funding of projects, and terms of contracts with customers and subcontractors are all impacted. The competitive advantages that an organization derives from reuse are improved time-to-market, improved knowledge of the domain of applications, recognition as a supplier of high-quality products, and improved productivity. Successful introduction of reuse requires an organization's software development process to be defined, that is, level 3 of the Capability Maturity Model scale [38]. In such cases, the conclusions from industrial reuse efforts [16, 50, 71] are typified by Hewlett-Packard's experience [53] of a 2-4 times return-on-investment from reuse. This is due to improved quality (up to 51% reduction in defects), improved productivity (up to 57% increase), and a 42% reduction in time-to-market. These net improvements come even though the development of components for reuse typically requires 20% more effort in each of the following phases: requirements analysis, module interface design, and coding.

The paper is divided into two main parts: the context or background for reusable object-oriented design — reuse, domain analysis, and design; and the reusable design artifacts themselves — software architectures, application frameworks, design patterns, and class libraries. The open problems are summarised in the conclusion.

We highlight domain analysis in section 2.2 because of its significance to systematic reuse and domain-dependent design artifacts such as application frameworks and class libraries. Reuse requires *understanding* the artifact to be reused, and an understanding of how to reuse the artifact. Hence, descriptions or models of design artifacts are important aspects of our presentation.

## 2 Context for Reusable Object-Oriented Design

This section provides a context for the research on reusable object-oriented design. In particular, we briefly provide background on reuse, domain analysis, and design. We assume the reader is familiar with the basics of the object-oriented paradigm [45] and software engineering [32].

### 2.1 Reuse

Reuse applies not only to source-code fragments, but to all the intermediate products generated during software development, including requirements, documents, system specifications, and design: indeed any information that the developer needs to create software [9, 27, 67, 64]. The reuse of domain-independent software components for an organization usually involves features common to all application systems. These include common data structures, graphical user interfaces, interfaces to databases, and networks. The components may be purchased as a commercial library, or developed in-house, and then reused in new applications.

It is important to separate the two sides of the reuse process: development of components **for** reuse and development of new applications **with** reuse of existing components. The first side produces reusable components, and the other side consumes them during the development of new applications. It is also important to state that while most reuse may be reuse of source code, the term *component* also includes designs, specifications, and requirements, all of which may also be reused. The main steps in development **for** reuse are

- to *re-engineer* components from existing systems to make them more general and reusable,
- to *qualify* the components to ensure they are of acceptable quality for the library, and
- to *classify* the components as to purpose and dependencies in order to facilitate subsequent retrieval from the library.

The main steps in development **with** reuse are

- to *retrieve* components from the library according to some need of the application under development,
- to *evaluate* the quality and appropriateness of the candidate components, and
- to *adapt* a component if it cannot be used as-is.

Tracz's 3C Model [73] of reusable software components indicates that at least the following information must be stored about each component, and used in its classification: **concept**, the abstraction captured in a component; **content**, the implementation of that abstraction; and **context**, the environment in which component is designed to operate. The quality factors of a component which affect reusability are the general ones of reliability, testability, and maintainability, and some which are more reuse specific quality factors: flexibility, portability, understandability, and confidence [68]. Confidence differs from reliability in that confidence is an estimate of the probability of error in unforeseen contexts, while reliability estimates the probability of error in a fixed context.

In development **with** reuse, there may be *direct* reuse of a component if it precisely matches the needs of the application. If the component must be adapted, then the following forms of adaptation might be possible:

- reuse by *instantiation*, where arguments to a parametrized component are supplied;
- reuse by *specialization*, where a subclass of a component is derived; or

- reuse by *adaptation*, where the textual source of a component may be freely edited.

Reuse by *composition* of the adapted components is the norm for domain-independent components. A typical example is using pipes to compose commands in Unix shell programs. Reuse in mature domains or organizations may be reuse by *generation*, where the component is generated automatically from a description of the desired solution. The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

### 2.1.1 Scope of Reuse

Three different scopes of reuse have been identified [11]:

**General reuse** is independent of the domain of application. It almost entirely reuses components such as common data structures and graphical user interfaces, that are found in most applications.

**Reuse within an application domain** restricts its focus to a single domain, such as insurance, or missile guidance. It is important to specify the common aspects of application systems for this domain, and to also specify the range of variability found in the domain. This is one role of *domain analysis*.

**Reuse within a product family** focuses on one product line within an application domain. A generic architecture is developed for this product family, so that there is a well-defined role for each component.

Reuse could be carried out within an organization, or be inter-organizational through the marketing of reusable components. Reuse within an organization has been profitable at each of the three scopes. Reuse is most successful when it is systematic and focuses on a narrow, well-understood application domain, or on a product family [27, 33].

## 2.2 Domain Analysis

Reuse which is specific to an application domain relies on an analysis of the domain to discover the common aspects of application systems for this domain, and to also discover the range of variability found in the domain. Domain analysis is

“a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems.” [74]

The domain is analyzed from the user’s and customer’s perspective (as in normal systems analysis), and also analyzed from the developer’s perspective to discover common software components, interfaces, and architectures [4, 74]. The sources of domain knowledge are the technical literature of the domain, requirements documents for current application software, surveys of customer needs, the architecture and components of existing systems, and domain experts. The results of domain analysis may include a taxonomy of the concepts in the domain and their relationships, which may be used to aid classification of components; standards concerning data formats, interfaces, etc; models of system architectures or components; and languages for describing domain entities. The last two outputs play significant roles in product-specific reuse.

The domain analysis identifies domain concepts, constructs, and code. Domain concepts may be reused during analysis, as the concepts capture the user- and customer-level perspective of the domain. Domain constructs are design artifacts such as architectural descriptions, abstract classes, or module interfaces, some of which are traceable to user-level concepts. The domain constructs may be reused during design. Domain code is actual code extracted from existing implementations, or specifically developed for reuse.

Source code is the implementation of a module, or class, and may be reused during the implementation phase. Each concept might correspond to several constructs, and each construct might correspond to several code fragments, which implement the construct or concept.

Domain analysis may identify common architectural features, and may develop a language to describe the variable aspects of the product family. A generic architecture is developed for this product family, so that there is a well-defined role for each component. The architecture, and its realization, is variously called a *reference architecture*, a *domain specific software architecture* (DSSA), or an *application framework* [16, 79]. The role of a component within the framework may also form part of the domain taxonomy and be used in classification and retrieval.

For mature architectures and product families, one can build an *application generator* [22]. This is reuse by generation, where the components or system are generated automatically from a description of the desired solution. The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

Not all domains lend themselves to reuse. For example, in a domain where difficult real-time constraints are very important, these performance considerations may always demand monolithic applications. Hence there is no possibility of separating out reusable components.

## 2.3 Design

Design [18] is the activity to produce a description of *how* to perform a task in such a way that it satisfies the customers' requirements and also satisfies any constraints imposed by the environment in which the system is to be used. Design methodologies [18, 26, 44] typically have three principal components:

- a *representation* of the design using one or more notations;
- a *process* for developing or transforming the representation; and
- a set of *heuristics* maybe guiding the order of steps, or the selection of issues that are to be addressed.

Heuristics are necessary since there is no pre-determined way to converge to a good design. Typically one postulates a tentative design and, by analysing its merits and deficiencies, the design is refined to a complete working design. Generally, an architectural design is developed before a detailed design. The architectural design describes the main subsystems and modules, their responsibilities, the division of control, and the interface or protocol of each module. The detailed design describes the internal mechanism of each module.

Design is an example of a “wicked problem” [18, p.19-21]. Often, when pursuing the resolution of an issue, the designer discovers other related issues of which the designer was not aware. The problem becomes more complicated as you approach a solution. Thus, the requirements for a system may not be fully understood until the design is complete. Design processes may be modelled [63] as an exploration of issues, the arguments for and against a position on an issue, and the design steps taken because a position has been adopted. Beck and Johnson [7] use this approach to describe an architecture by the issues encountered and the design patterns chosen to address each issue.

Parnas' writings [60, 61, 62] have lead to the consensus that “design for change” is an overarching principle for software design [32, p.65], and that information hiding is an essential step to achieving design for change. The object-oriented paradigm stresses information hiding through the use of private state variables for objects and loose coupling through message passing [45], so it is well-suited to design for change.

The *representation* or *model* of the design forms the basis of communication, analysis, and documentation. A model and the views of a model are described using notations. Budgen [18] classifies design notations into three categories:

**diagrammatic** notations based on a visual graphical representation;

**textual** notations written in pseudocode or natural language; and

**mathematical** notations based on logic, algebra, set theory, or other mathematical notations.

There are generally multiple views of a system. Each view focuses on a single aspect of the system in an attempt to reduce complexity. Object-oriented design methodologies, for example, commonly use the following viewpoints or perspectives [13, 18, 66]:

**structural** viewpoints of the static aspects of a system, which may include the organization of subsystems and modules at the architectural level;

**functional** viewpoints which seek to describe what the system does in terms of its tasks;

**behavioral** viewpoints describing the dynamic or causal nature of events and system responses during execution; and

**data-modeling** schemas, or class diagrams, concerned with the data used within the system and the relationships between these.

The notations used for these viewpoints also support *decomposition* or *hierarchies* to further assist humans to control the complexity of even a single aspect of a system. For example, a developer might use nested modules to describe the structural view of a system; hierarchical data flow diagrams to describe functionality; statecharts, which provide nesting, abstraction, and decomposition, to model behavior; and a class inheritance hierarchy to model data [18]. Each viewpoint should be derived from a consistent underlying model. Harel [35] is optimistic that developers can exercise greater intellectual control over complexity by using models throughout the development process. Harel is a supporter of multiple models, which have both a visual representation and a solid semantic foundation. A visual model aids human cognitive skills; a semantic foundation allows analysis of the models, such as providing answers to “what-if” questions.

### 3 Software Architectures

Software architectures provide an abstract description of the organisational and structural decisions that are evident in a software system.

“Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

This is the *software architecture* level of design.” [31]

One important aspect of this area is compiling a *catalogue* of existing software architectures. The primary reference is the work of Mary Shaw and her colleagues [31]. Their list of architectural styles follows.

- *Pipe-and-Filters* architectures, like that supported by the Unix shell, connect filters in a linear fashion. Each filter has one stream of inputs and one stream of outputs.
- *Data Abstraction* and *Object-Oriented Organization* architectures promote the decomposition of the system into entities (data type variables or objects) that encapsulate their implementation details and present an interface that completely describes their behaviour or functionality.
- *Event-based, Implicit Invocation* architectures are based on components that register an interest in (a class of) events. The components are invoked in response to the occurrence of an event implicitly rather than being called directly by another component.
- *Layered System* architectures are organized as a hierarchy of layers, each layer providing services to the layers above it and each layer being a client for the services provided by layers below it.
- *Repository* architectures have distinct components for a central data store (the repository) and those components which operate upon the repository.
- *Table Driven Interpreter* architectures implement a software virtual machine (the interpreter) by separating the interpretation engine from a table that describes the machine behaviour.
- Heterogeneous architectures combine architectural styles.

The development of a catalogue of architectures would be greatly assisted if corporations released details of their system architectures: information which is now often proprietary.

The ultimate aim of their research is a *taxonomy* of architectures to organize our knowledge by describing common and distinguishing features of the architectures. This may help designers to appreciate the breadth of choices and trade-offs, and may guide the discovery or invention of new artifacts. It may also assist the development of design notations and languages, which could be incorporated in development environments specific to individual architectural styles [30].

The book on OMT [66] also contains very useful sections describing architectures and their use in system design. There is good general agreement of their list below with the above list of architectures.

- *Batch Transformation* architectures transform the entire input data set once.
- *Continuous Transformation* architectures transform the input data continuously in response to incremental changes in the input.
- *Interactive Interface* architectures are dominated by external interactions.
- *Dynamic Simulation* architectures simulate evolving real-world objects.
- *Real-time System* architectures are dominated by strict timing constraints.

- *Transaction Manager* architectures process transactions on data stores that are being accessed in a concurrent and distributed fashion.
- Hybrid systems combine architectural styles.

There is a strong relationship between the study of software architectures and the study of programming at the architectural level that is pursued in megaprogramming [78] and the design of module interconnection languages [65].

### 3.1 Reusing Software Architectures

Rumbaugh et al [66] provide a methodology for applying an architecture that is based on

- *characterising* the kinds of systems to which the architecture is applicable;
- presenting important *design principles* which must be observed when applying the architecture; and
- providing a sequence of *detailed steps* to follow when developing a design based on the architecture.

we illustrate this methodology for the Interactive Interface Architecture in Section 3.1.1.

Another approach to utilising architectural styles is to encode a description of the components and constraints of an architecture into an software development environment. Thus the environment forces the designer to conform to the specified style. Garlan et al [1, 30] describe an architecture in terms of its components, connectors, configurations, ports, and roles. Hence a description of an architectural style provides

- a *vocabulary* of the basic design elements (components and connector types),
- a set of *configuration rules* which constrain how components and connectors may be configured,
- a *semantic interpretation* which defines when suitably configured designs have a well-defined meaning as an architecture, and
- *analyses* that may be performed on well-defined designs.

From such a description, their system, Aesop, can generate a development environment that is tailored to the given architectural style.

#### 3.1.1 Example: Interactive Interface Architecture

This section summarises the methodological information of Rumbaugh et al[66, section 9.10.3] for applying the interactive interface architectural style.

**Characterisation** An interactive system is dominated by the interactions with external agents, such as users, sensors, and IO devices, over which it has no control. So events and interactions occur independently of the system, although the interactions may be in response to system prompts or requests.

**Design Principles** Isolate the interactive interface from the rest of the system. Understand the sequence or syntax or protocol of the interaction. The dynamic model of behaviour is very important.

### Steps in Designing an Interactive Interface

- Isolate the objects which form the interface from the objects that define the semantics of the application.
- Use predefined objects to interact with external agents, if possible. For example, a toolbox like XR11 or Motif.
- Use the dynamic model to structure the program. Use concurrent control (multi-tasking), or use event-driven control (interrupts or callbacks).
- Isolate physical events from logical events.
- Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.

### 3.2 Other Information

Design patterns can be used to describe an architecture and document the rationale for its development [7]. Architectural styles can be used to tailor proof techniques or provide a framework for modular proofs. Layered architectures can be reasoned about using modular proofs for each layer and then relying on the composition theorem of Lam and Shankar [49] to show that the system as a whole has the desired properties. The equivalence of two architectures may be proven [59] modulo the equivalence of their components, thus providing assurance about a system re-structuring or about a refinement from an abstract architecture to a concrete implementation of that architecture. Moriconi and Qian [59] illustrate their theory by proving the equivalence between a pipe-and-filter architecture for a compiler and one which uses an abstract data type for the parse tree.

## 4 Application Frameworks

For a given application domain, the process of domain analysis can draw from existing software applications to identify common architectural features, and to describe their variable aspects. A generic architecture may be developed for a product family or application domain, so that there is a well-defined role for each component. A realization of the architecture is an *application framework*:

“a collection of abstract classes, and their associated algorithms, constitute a kind of *framework* into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. The framework consists of the abstract classes, the operations they implement, and the expectations placed on the concrete subclasses” [25, p.92]

“A framework is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific.” [42, p.25]

A framework is usually designed by experts in a particular domain and it is used by non-experts. It allows the user to reuse abstract designs, and pre-fabricated components in order to develop a system in the domain. A user may also customize existing components by subclassing. The design of the framework incorporates decisions about the distribution of control and responsibility, the protocols followed by components when communicating, and implementations for each of the major algorithms. Often the implementations are template methods that embody the overall structure of a computation and that call user’s classes to perform sub-steps of the algorithm. Default implementations of each user class may be provided, and the user will subclass in order to override or specialize the operation which implements the sub-step.

A good characterization of the relationship between a framework and user’s classes is “Don’t call us, we’ll call you.” So the classes defined in the framework call the user’s code, whereas the traditional use of class libraries is for the user’s code to call the library classes.

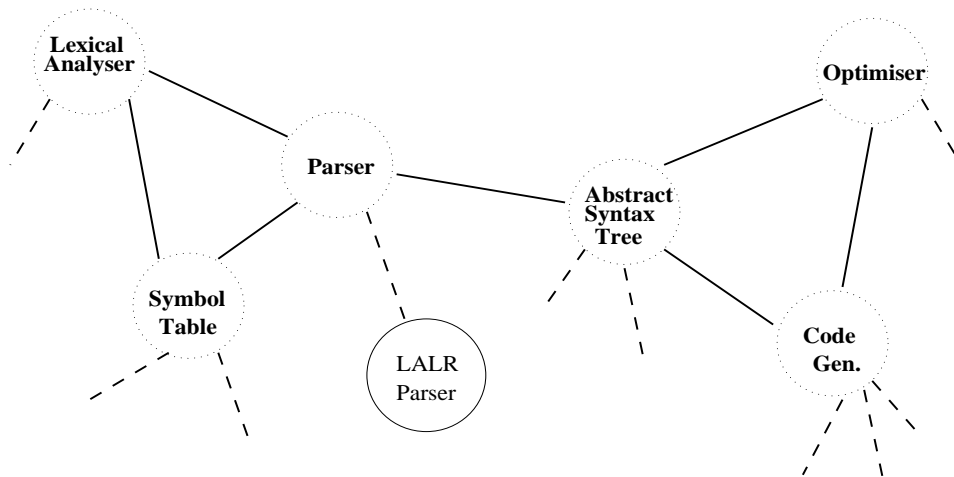


Figure 1: Compiler Framework

For example, a framework for program translation or compilation would provide an abstract class for each of the roles identified: **Lexical Analyser**, **Parser**, **Symbol Table**, **Abstract Syntax Tree**, **Optimiser** and **Code Generator** shown in Figure 1. Furthermore, there would be a class or main program orchestrating the overall communication and control of the compilation process. Concrete subclasses, such as **LALR Parser** would implement specific algorithms and/or representations for a role. A reuser of the framework

would select a concrete subclass for each role, if a suitable one existed in the associated class library, or create their own subclass (usually by specializing an existing one. Once the selected configuration is compiled, a new instance of the application is ready for use.

Early examples of application frameworks were for *graphical user interfaces* (GUI), including MACAPP [3], and INTERVIEWS [54]. There is now an abundance of GUI frameworks that have been used successfully on many platforms ranging from DOS to UNIX, such as MACAPP for MacIntosh, OWL-WINDOWS for DOS/WINDOWS, and MOTIF for UNIX. Frameworks now exist for a broad range of application domains such as ET++, an *editor* toolkit [77] which has recently been used in MET++ which is a framework for *multimedia* applications; RTL framework [43] for code optimization in compilers; CHOICES for object-oriented operating systems [19]; BEE++ for analyzing and monitoring distributed programs [17]; and others for network management and telecommunications [8], and financial engineering [10]. Don Batory[5] has developed the GENESIS framework for relational database systems, where a database is a composition of functional layers (or realms), and the framework consists of the realms, their type constraints as functions, and the alternative implementations. Commercially available (non-GUI) frameworks include SUPERTEVIEW for word processing, NEO.ACCESS for object-oriented databases, and NEXTSTEP MUSIC KIT for music and sound.

## 4.1 Developing a Framework

An application framework evolves in response to feedback from reusers. An initial framework is based on past experience or by careful construction of one or two applications, keeping in mind the need for flexibility, reusability and clarity of concepts. Each consequent reuse points out shortfalls in these qualities in the existing framework as one stretches the architecture to accommodate the new application. By addressing the issues raised, the framework evolves, gaining flexibility, coverage of domain concepts, and clarity of the concepts and the dimensions along which they vary.

The major steps in developing an application framework can be summarized as follows [42, 72]:

1. Identify and analyze the application domain and identify the framework. If the application domain is large, it should be decomposed into a set of possible frameworks that can be used to build a solution. Analyze existing software solutions to identify their commonalities and the differences.
2. Identify the primary abstractions. Clarify the role and responsibility of each abstraction. Design the main communication protocols between the primary abstractions. Document them clearly and precisely.
3. Design how a user interacts with the framework. Provide concrete examples of the user interaction, and provide a main program illustrating how the abstract classes are related to each other and to the classes for user interaction.
4. Implement, test, and maintain the design.
5. Iterate with new applications in the same domain.

The design and implementation of frameworks relies heavily on abstract classes, polymorphism (both parametric and inclusion polymorphism), and inheritance.

When analyzing existing applications to determine reusable components and abstractions, one might re-structure the classes in order to separate what is common across applications from what is unique to one application. Johnson and Foote [42] provide a set of rules to this end: designing reusable classes. They can be summarized as follows:

### Rules for Finding Standard Protocols

- Rule 1: Recursion introduction
- Rule 2: Eliminate case analysis
- Rule 3: Reduce the number of arguments
- Rule 4: Reduce the size of methods

### Rules for Finding Abstract Classes

- Rule 5: Class hierarchies should be deep and narrow
- Rule 6: The top of the class hierarchy should be abstract
- Rule 7: Minimize access to variables
- Rule 8: Subclasses should be specializations

### Rules for Finding Frameworks

- Rule 9: Split large classes
- Rule 10: Factor implementation differences into subcomponents
- Rule 11: Separate methods that do not communicate
- Rule 12: Send messages to components instead of `self` (in Smalltalk) or `this` (in C++)
- Rule 13: Reduce implicit parameter passing

The components of design patterns (described in the next section) also provide a goal or target for the re-structuring of classes, and may assist in determining some roles of the abstractions identified.

## 4.2 Describing Frameworks

Only a small amount of work[36, 37, 41] has been done on documenting, specifying, and reasoning about frameworks. The frameworks under consideration are chosen from toolkits for user interfaces and drawing programs. Only the Contracts paper[36] considers verifying correctness, but the authors offer no evidence of actual reuse which has benefited from their contracts. On the other hand, patterns[41] have been an important aspect of much actual reuse. There the emphasis is on documentation rather than specification, and certainly there is no concern for verification of correctness.

The documentation of a framework is very important to both the re-user of the framework, and to the developer/maintainer of the framework. These two audiences have different requirements:

**Documentation for reuse** illustrates how to customize the framework, often through examples, for typical reuses. It discusses which classes to subclass, and which methods to override, and whether combinations of classes and methods need to be specialized in unison to maintain a protocol of collaboration amongst the classes. It is prescriptive.

Johnson [41] introduced an informal *pattern language* that can be used for documenting a framework in a natural language. The documentation of a framework consists of a set of patterns where each pattern describes a problem that occurs repeatedly in the problem domain of the framework, and also describes how to solve that problem. Each pattern possesses the same format. The elements of a pattern are: description of its purposes, explanation of how to use it, description of its design, and some examples.

**Specification for reuse** is generally more descriptive than prescriptive: the re-user is left to figure out the implications of the specification in terms of the desired customization. The main concerns are to clearly specify the obligations on a user-defined subclass, any protocols which the user can customize, and the collaborations amongst classes that must be supported by the user-defined subclasses.

**Documentation and specification for general understanding** primarily assists the evolution of the framework. Traditional techniques for modules, such as the Larch family of interface languages [34], can be used for describing class interfaces and extended to include the obligations on subclasses [20]. The Eiffel language [57] supports design-by-contract through the declaration of assertions, preconditions, and postconditions.

Contracts [36, 37] are a high-level (abstract) construct for explicitly specifying behavioral composition, the obligations on participating objects, and interactions among groups of objects. A contract specifies a set of communicating participants and their contractual obligations which extend the signature of types and functions used in a class to include constraints on behavior that capture the behavioral dependencies between objects. A contract specifies preconditions on participants required to establish the contract, and the invariant to be maintained by these participants.

Two constructs for modifying a contract are provided: *refinement* and *inclusion*. Refinement and inclusion are used for deriving a class based on particular subclasses. This concerns the aspect of inclusion polymorphism that is necessary in building an application framework.

Contracts are refined by either specializing the type of a participant, extending its actions, or deriving a new invariant which implies the old. Consequently, the refinement of a contract specifies a more specialized behavioral composition.

The behavior of a subset of the participant in a large composition can be specified by means of simpler *sub-contracts*. These sub-contracts can simply be included in a large contract. The invariant clauses of a contract with sub-contracts are formed from the union of the corresponding statements from the sub-contracts.

### 4.3 Application Generators

For mature architectures and product families, one can build an *application generator* [22]. This is reuse by generation, where the components or system are generated automatically from a description of the desired solution. The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

“toolkit is a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications.” [42, p.26]

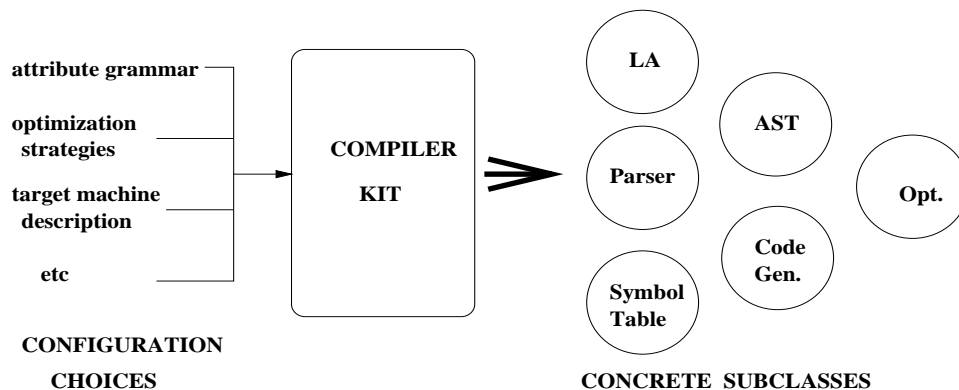


Figure 2: Toolkit for the Compiler Framework

Figure 2 shows possible inputs and outputs — the concrete subclasses for the compiler framework — of a compiler generator.

## 4.4 Other Information

Section 3.2 on software architectures is also relevant to application frameworks.

Batory and O'Malley [5] introduce the GENVOCA approach to designing and implementing hierarchical software systems with reuse based on the object-oriented paradigm. It aims at developing a domain-independent model of hierarchical software system design and implementation based on interchangeable software components and large-scale reuse.

The basic notions in this approach are *realms* and *components*. A *component* is a closely-knit cluster of classes that acts as a unit. Every component has an interface and an implementation. A component can be thought of as an object or as the implementation of a class. A *realm* is an abstraction of a set of components. A set of components possessing a common behavior is described at an abstract level by a realm. Consequently, every component is a member of a particular realm. The interface of a realm consists of the set of one or more classes including their objects, functions, and interrelationships. The interface of a component has additional information, such as source and object files. When designing a large system in a particular application domain, realms can be used as reusable design artifacts. The implementation of the system can then be accomplished by reusing the corresponding components. The language P++ [69] supports the GENVOCA approach.

## 5 Design Patterns

A pattern is a solution to a problem in a context [21, p.93]. A pattern is an abstract model of the problem and its solution. So the pattern expresses the relationship among the elements in a problem, and describes the context: it lets the designer focus on the abstract level (building block) when thinking about the solution rather than focusing on the low concrete level (bricks) [2, 52]. The pattern mechanism also describes quality, impact and alternatives.

“Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities.” [28, p.407]

“Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” [29, p.3].

Design patterns capture design experience at the micro-architecture level, by specifying the relationship between classes and objects involved in a particular design problem. By documenting, cataloging and classifying this experience, design patterns allow it to be reused. Since design patterns only record proven reusable design techniques, they offer improved reusability and reliability for developers of new systems. Furthermore, design patterns can improve the documentation and maintenance of existing systems through the reuse of the explicit specification of class and object interactions and their underlying intent [29]. So design patterns constitute a reusable base of experience for building reusable software.

Design patterns tend to be independent of the application domain since they are smaller and less specialized than frameworks and architectures. They are abstract since they describe a pattern (of structure or collaboration) and not a concrete instance of the pattern. While concrete examples might form part of the documentation of a design pattern, the cataloguer has already abstracted a higher level description from these examples.

Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. The vocabulary of patterns — their names, the roles played by participants, and the names of collaborations — enrich the vocabulary of object-oriented designers forming a part of their language when communicating with colleagues and when presenting their work to others.

A design pattern provides an abstraction above the level of classes and objects. This allows a designer to work at a more abstract level and to reduce system complexity. Each design pattern identifies a design problem, constraints, solution to that problem and other alternatives, meanwhile it encapsulates a well-defined problem/solution [52, p.42] as a building block for constructing more complex designs.

Design patterns help reduce the required learning time for a library when the library uses the common vocabulary of roles in a design pattern. The library user is then familiar with the terminology in the documentation and knows from the roles how the class should be reused.

A catalogue of design patterns offers examples of good design, and provides proven solutions to design problems in a well-organized form. This can help a novice learn design skills more quickly and to perform more like an expert.

Design patterns provide a target for the reorganization or refactoring of class hierarchies.

### 5.1 Documenting Design Patterns

The purpose of design patterns is to reuse design. Clearly the problem, the context, and the solution are documented. However, besides the description of the structural aspects of a pattern, it is necessary to also record the information pertinent to the critical issues considered during design work, such as design decisions, alternatives and trade-offs. In general, a design pattern has four essential elements:

- 1) Name(s) of Pattern — to identify a design problem, and to produce terminology for thinking and communicating.
- 2) Problem Explanation — to describe a set of pre-conditions which should be met to apply the pattern.
- 3) Solution Description — to describe the relationships, responsibilities and collaborations among the elements involved in the pattern.
- 4) Consequences Analysis — to present the results and trade-offs of applying the pattern.

Typically this information is provided in a template, though some aspects of Contracts [36] for documenting frameworks actually document patterns: the connection is elaborated in work of Lajoie [47, 48]. The “standard” way to document a design pattern is the template [29, p.6] shown in Figure 3. A template provides a consistent format for documentation which makes design patterns easier to learn, compare and use.

Design patterns vary in their granularity and level of abstraction, as well as in what they actually do. Design patterns are **classified** by two criteria [29, Table 1.1, p.10]: **purpose** which can be *creational*, *structural*, or *behavioral*; and **scope** which can be *class*, *object* or *compound*. Purpose indicates the outcome of applying the pattern: Does it organize how classes or objects are created? Does it provide a useful structure amongst classes or objects? Does it provide a useful dynamic behavior? Scope indicates the granularity; that is, whether the pattern mainly works at the level of a class, an object, or a number of objects.

These are early days for design patterns, and much work is being done on ways of documenting and classifying patterns [75].

## 5.2 Reusing Design Patterns

Design patterns support issue-based design by providing a range of solutions for certain issues that commonly arise in object-oriented design. Generally these are issues of providing flexibility so that some aspect of the design can change easily. Knowing what varies, one can access related design patterns [29, Table 1.2, p.30]. It is then a matter of understanding the abstraction of the design pattern and matching components of the actual design to the abstract participants of the pattern.

Beck and Johnson [7] illustrate the use of design patterns in developing the design for HOTDRAW, a framework for drawing editors. This is actual a re-documentation of an existing design [41], but it very clearly shows the nature of issue-based design.

In [29, pp.33–77], a case study is presented in the design of a “What-You-See-Is-What-You-Get” document editor called LEXI. The document can mix text and graphics freely in variety of formatting styles, with pull-down menus and scroll bars, and a collection of page icons for jumping to a particular page in the document. Eight design patterns are illustrated in the case study.

## 5.3 Other Information

Patterns are a way of developing and packaging reusable software components [6]. *Abstractors*, who create reusable pieces, and *elaborators*, who massage those pieces to fit the needs of a user, are introduced as two sets of players in the world of software development. A Multicurrency Library, as an example, is described to illustrate how patterns can be used as “a way for abstractors to communicate their intent.”.

The first step to develop a pattern is a process of discovery [21, p.94]. “Beyond its elements, each building [or town] is defined by certain patterns of relationships among the elements.... These relationships are not extra, but necessary to the elements.... The elements themselves are patterns of relationships.” [2, 21] “And finally, the things which seem like elements dissolve, and leave a fabric of relationships behind,

---

**Design Pattern Name and Classification**

The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

**Intent**

What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

**Also Known As**

Other well-known names for the pattern, if any.

**Motivation**

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help the reader understand the more abstract description of the pattern that follows.

**Applicability**

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

**Structure**

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [66]. The interaction diagrams [13] is applied to illustrate sequences of requests and collaborations between objects.

**Participants**

Describe the classes and/or objects participating in the design pattern and their responsibilities.

**Collaborations**

Describe how the participants collaborate to carry out their responsibilities.

**Consequences**

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

**Implementation**

What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

**Sample Code**

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

**Known Uses**

This section presents examples from real systems. We try to include at least two examples from different domains.

**Related Patterns**

What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

---

Figure 3: Documentation Template for Design Patterns

which is the stuff that actually repeats itself, and gives the structure to a building or a town.” [2, 21] To find a pattern among some low-level elements, such as classes and objects, means to find the relationships among them in a context. A pattern results from the experience of being an elaborator several times. A pattern is problem-oriented, not solution-oriented, so the developer is looking for the issues or tensions that the pattern resolves.

## 6 Class Libraries

There have been many class libraries developed, and their design has been extensively documented in the literature [42, 46, 56, 58, 70], so this section will be brief.

Object-oriented class libraries embody

- the design of interfaces,
- the design of a class hierarchy, and
- the design of policies or conventions, such as those on reporting errors in calls to library classes.

The most effective libraries are domain-specific and often complement an application framework by providing a choice of concrete classes to instantiate the abstract classes in the framework [42, p.25]. Korson and McGregor [46] list the following necessary attributes of a reusable object-oriented class library:

**complete** general model for each concept covered by the library;

**consistent** definitions and naming throughout the library;

**easy-to-learn** because of the organization of the documentation, design, and implementation;

**easy-to-use** because the necessary classes are easy to find, understand, and use;

**efficient** implementations of each algorithm, and the selection of appropriate algorithms;

**extendable** classes via specialization or composition;

**integrable** with other libraries;

**intuitive** to those familiar with the standard practice for the domain covered by the library;

**robust** classes that deal reasonably with erroneous arguments to calls and with erroneous sequences of calls; and

**support** from the vendor.

The rules of Johnson and Foote [42] provide some guidance as to the evolution of a reusable class for a library, and Meyer [58] illustrates clarity of concepts, the precise style of Eiffel interface contracts, and just how few arguments the methods in a reusable class should have. The design of reusable class libraries is an iterative process requiring a clear vision of the domain of the library and also requiring much feedback from actual reusers about the classes, their interfaces, and the documentation.

Object-oriented languages generally have a standard library providing basic mathematical objects, I/O files and streams, and utility classes such as *String*.

Class libraries for common data structures are specific to the domain of data structures, and are the result of a long history of studying abstract data types and collection classes. The development of the Booch Components, first in Ada [12] then in C++ [14, 15], clearly uses a domain analysis.

Utility class libraries for persistence, threads, and distributed computing follow a set of conventions, as do classes for I/O files and streams.

User interface libraries complement the corresponding application framework and must be consistent with the conventions and division of responsibility and control of that framework.

## 7 Conclusions

There are some obvious conclusions from our survey of reusable object-oriented design. The most apparent is that both fields of study — software reuse and software design — are under active development, and while we may be approaching a mature understanding of those fields individually, we have hardly begun to understand their intersection which includes reusable object-oriented design. One outstanding issue in software reuse is *understandability* of artifacts: how does one document, specify, classify, and annotate artifacts so they are easily understood by a potential reuser. Two outstanding issues for software design are how to document the rationale or reasoning behind a design — especially the false leads rejected during the design process — and metrics relevant to the qualities which a design should possess.

The major problems of software development **with** reuse of retrieval and understanding exist for design artifacts as they do for code artifacts.

One clear conclusion is that reusable design artifacts are the result of evolution and iteration. This is especially true for application frameworks and class libraries — the concrete artifacts, where all details must be spelt out.

Another clear conclusion is that models and notations play several important roles. The abstract artifacts are themselves models, but even for them the role of models and notations extends much further.

- Models and notations are an aid to *communication*.
- Models and notations improve *understanding* through precision, conciseness, and visual cues.
- A model or documentation template often provides a *checklist* of the information needed to fully understand or reuse a design artifact. For example, these may include:
  - responsibilities of each participant;
  - collaborations amongst participants;
  - purpose of the artifact; and
  - preconditions for applicability.

There are many open problems for architectures, frameworks, micro-architectures, and, to a lesser degree, for the design of class libraries. We have broadly categorized these into problems of retrieval, understanding, and evaluation.

**Retrieval** problems are those related to the classification and description of design artifacts, as well as the problem of enlarging our catalogue of known reusable design artifacts.

Each of the following is aimed towards enlarging and organizing our knowledge of design artifacts. This provides a base from which to retrieve design artifacts and also provides a conceptual classification scheme that forms the basis of the attributes mentioned in retrieval queries. Although we have seen examples of work on each of these areas in this survey, there is much more that needs to be done still.

- A *taxonomy* of design artifacts provides a map of the space of artifacts, their commonality and their differences. This organizes our knowledge, helps designers to appreciate the breadth of choices and trade-offs, and may guide the discovery or invention of new artifacts, and assist the development of design notations and languages.
- *Classification* of individual artifacts according to their significant or distinguishing features aids our understanding of that individual artifact and contributes to the development of the taxonomy.
- *Cataloguing* the existing design artifacts requires assistance from industry to release details of the design of their systems. Our understanding of software architectures and application frameworks depends heavily on enlarging the base of examples in the catalogue, particular to assist the development of a taxonomy. The invention and discovery of new designs is also a part of this open problem.

**Understanding** problems are those related to reducing the effort by a reuser to understand a design artifact and to understand how to reuse a design artifact.

- *Documentation styles and standards* are required to ensure that all information needed by reusers is documented, and to present the information to reusers in a timely fashion: that is, precisely when they need it.

The spiral approach of patterns [41] tailors the timing and volume of information that a reuser of a framework is presented with. The documentation templates for design patterns [29] ensures that each item of necessary information is provided, and that examples of use are also included.

Documentation of architectures and class libraries is still in need of good styles and standards.

- *Formal specification and description* of design artifacts claim to provide improved precision, conciseness, and reasoning. Each of which may improve understanding. Most of the work on the description of design artifacts has been done by practitioners who do not share a belief in the merit of formal specifications: they used natural language and diagrams for their documentation. As a consequence there is only a small body of work on how to formally specify design artifacts [1, 20, 23, 24, 36, 37, 59]. There is scope for further work in this area.
- *Experimental validation* of claims that certain documentation or specification styles do improve understanding in practice is required. All aspects of research into software development need a firm experimental foundation.

**Evaluation** problems include those of comparing design artifacts, those concerned with the evolution of artifacts, and the problems of evaluating design artifacts.

- *Metrics for design artifacts* are in very short supply. The correlation between metrics and the desirable qualities of a design also needs to be established quantitatively. The metrics research community is large and active so we predict that a stream of new design metrics will be forthcoming. There is also a strong interest from practitioners who are establishing quality assurance programmes and are collecting the necessary data to confirm any correlation between measurement and quality.
- Specifying and describing the *evolution of a design* and the differences between designs may require a notion of a “design delta” to capture the increment of change or difference. Alternatively, evolution could be viewed transformationally, and the increment would be a description of the transformation. Ralph Johnson and his students call these transformations “refactorings”. Given the importance of evolution, refinement, and iteration to design and reuse, this is a major open problem for reusable object-oriented design.

**Acknowledgements:** This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and *Fonds pour la Formation de Chercheurs et l’Aide à la Recherche*.

## References

- [1] G. Abowd, R. Allen, D. Garlan, *Using style to give meaning to software architecture*, Proceedings of SIGSOFT'93: Foundations of Software Engineering, ACM Software Engineering Notes **18**, 3 (December 1993) 9–20.
- [2] C. Alexander, **A Timeless Way of Building**, Oxford University Press, 1979.
- [3] Apple Computer, **Macapp 2.0 General Reference Manual**, 1990.
- [4] G. Arango and R. Prieto-Diaz, *Domain analysis: Concepts and research directions*, in **Domain Analysis: Acquisition of Reusable Information for Software Construction**, R. Prieto-Diaz and G. Arango (eds), Computer Society Press Tutorial, May 1989.
- [5] D.S. Batory and S.W. O'Malley, *The design and implementation of hierarchical software systems with reusable components*, ACM Trans. on Software Engineering and Methodology **1**, 4 (1992) 355–398.
- [6] K. Beck, *Patterns and software development*, Dr Dobbs Journal **19**, 2 (1993) 18–23.
- [7] Kent Beck and Ralph Johnson, *Patterns generate architectures*, ECOOP'94.
- [8] Roger P. Beck, Satish R. Desai, Doris R. Ryan, Ronald W. Tower, Dennis Q. Vroom, Linda Mayer Wood, *Architecture for large-scale reuse*, AT&T Technical Journal **71**, 6 (1992) 34–45.
- [9] Ted J. Biggerstaff and Alan J. Perlis, *Software Reusability*, volume I—Concepts and Models; volume II—Applications and Experience; ACM Press/Addison-Wesley, 1989.
- [10] A. Birrer and T. Eggenschwiler, *Frameworks in the financial engineering domain: An experience report*, **ECOOP'93 — Object-Oriented Programming**, O.M. Nierstrasz (ed.), Springer-Verlag, Berlin, 1993, pp.21–35.
- [11] Blandine Bongard, Björn Grönquist, Danielle Ribot, *Impact of reuse on organizations*, Proc. Reuse'93, Lucca, Italy, 24–26 Mar. 1993 (IEEE CS Press)
- [12] Grady Booch, **Software Components with Ada — Structures, Tools, and Subsystems**, Benjamin/Cummings, Menlo Park, CA, 1987, (xx)+635 pages.
- [13] Grady Booch, **Object-Oriented Design with Applications**, 2nd edition, Benjamin/Cummings, 1993.
- [14] Grady Booch and Michael Vilot, *The design of the C++ Booch components*, ECOOP/OOPSLA 90, ACM SIGPLAN Notices **25**, 10 (1990) 1–11.
- [15] Grady Booch and Michael Vilot, *Simplifying the Booch components*, The C++ Report (June 1993) pp.41–52.
- [16] Christine L. Braun, *Reuse*, in **Encyclopedia of Software Engineering**, John Marciniak (ed.), John Wiley and Sons, 1994, pp. 1055–1069.
- [17] B. Bruegge, T. Gottschalk, B. Luo, *A framework for dynamic program analyzers*, **OOPSLA'93**, pp.65–82.
- [18] David Budgen, **Software Design**, Addison-Wesley, 1993.
- [19] Roy H. Campbell, Nayeem Islam, David Raila, Peter Madany, *Designing and implementing CHOICES: An object-oriented system in C++*, Communications ACM **36**, 9 (September 1993) 117–126.

- [20] Yoonsik Cheon and Gary Leavens, *A quick overview of Larch/C++*, Technical Report **93-18**, Dept of Computer Science, Iowa State University, June 1993, 34 pages.
- [21] P. Coad, *Patterns (Workshop)*, *OOPSLA'92*, OOPS Messenger **4**, 2 (October 1993) 93–96.
- [22] J. Craig Cleaveland, *Building application generators*, IEEE Software **5** (July 1988) 25–33.
- [23] M. Cline and D. Lea, *The behavior of C++ classes*, Proc. of the Symposium on Object-oriented Programming Emphasizing Practical Applications, Marist College, 1990.
- [24] W.R. Cook, *Interfaces and specifications for the Smalltalk-80 collection classes*, **OOPSLA'92**, ACM SIGPLAN Notices **27**, 10 (October 1992) 1–15.
- [25] L. Peter Deutsch, *Reusability in the Smalltalk-80 programming system*, (pp. 72–76, ITT Proceedings of the Workshop on Reusability in Programming 1983) in **Tutorial: Software Reusability**, Peter Freeman (ed.), IEEE Computer Society Press, 1987, pp.91–95.
- [26] R.G. Fichman and C.F. Kemerer, *Object-oriented and conventional analysis and design methodologies*, IEEE Computer **25**, 20 (1992) 22–39.
- [27] W.B. Frakes and S. Isoda, *Success factors of systematic reuse*, IEEE Software **11** (September 1994) 15–19.
- [28] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, ECOOP'93, O.M. Nierstrasz (ed.), Lecture Notes in Computer Science **707**, Springer-Verlag, 1993, 406–431.
- [29] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [30] D. Garlan, R. Allen, J. Ockerbloom, *Exploiting style in architectural design environments*, Proceedings of SIGSOFT'94: Foundations of Software Engineering, ACM Software Engineering Notes **19**, 5 (December 1994) 175–188.
- [31] David Garlan and Mary Shaw. *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Company, 1993.
- [32] C. Ghezzi, M. Jazayeri, and D. Mandrioli, **Fundamentals of Software Engineering**, Prentice-Hall, 1991.
- [33] M.L. Griss, *Software reuse: From library to factory*, IBM Systems Journal **32**, 4 (1993) 548–566.
- [34] John V. Guttag and Jim J. Horning, with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing, **Larch: Languages and Tools for Formal Specification**, Springer-Verlag, New York, 1993.
- [35] David Harel, *Biting the silver bullet: Toward a brighter future for system development*, IEEE Computer **25**, 1 (January 1992) 8–20.
- [36] R. Helm, I.M. Holland, and D. Gangopadhyay, *Contracts: specifying behavioral compositions in object-oriented systems*, ACM SIGPLAN Notices **25** (1990) 169–180.
- [37] Ian M. Holland, *Specifying reusable components with contracts*, **ECOOP'92**, Lecture Notes in Computer Science **615**, Springer-Verlag, 1992, pp.287–308.
- [38] Watts Humphrey, **Managing the Software Process**, Addison-Wesley, 1989.
- [39] Michael A. Jackson, **Principles of Program Design**, Academic Press, New York, 1975.
- [40] Michael A. Jackson, **System Development**, Prentice-Hall, 1983.

- [41] R.E. Johnson, *Documenting frameworks using patterns*, ACM SIGPLAN Notices **27** (1992) 63–76.
- [42] R.E. Johnson and B. Foote, *Designing reusable classes*, Journal of Object-Oriented Programming **1** (1988) 22–35.
- [43] R.E. Johnson, C. McConnell, J.M. Lake, *The RTL system: A framework for code optimization, Code Generation — Concepts, Tools, techniques*, R. Giegerich and S.L. Graham (eds), Springer-Verlag, London, 1991, pp.255–274.
- [44] C.B. Jones, **Systematic Software Development using VDM**, Prentice-Hall, 1986.
- [45] Tim Korson and John D. McGregor, *Understanding object-oriented: A unifying paradigm*, Communications of the ACM **33**, 9 (September 1990) 40–60.
- [46] Tim Korson and John D. McGregor, *Technical criteria for the specification and evaluation of object-oriented libraries*, Software Engineering Journal (March 1992) 85–94.
- [47] Richard Lajoie, **Using, Reusing and Describing Object-Oriented Frameworks**, Master Thesis, McGill University, Montreal, July 1993.
- [48] Richard Lajoie and Rudolf K. Keller, *Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert*, Colloquium on Object-Orientation in Databases and Software Engineering, 62nd Congress of ACFAS, Montreal, May 16–17, 1994.
- [49] Simon S. Lam and A. Udaya Shankar, *A theory of interfaces and modules I — Composition theorem*, IEEE Trans. Software Eng. **20**, 1 (January 1994) 55–71.
- [50] R.G. Lanergan and C.A. Grasso, *Software engineering with reusable designs and code*, IEEE Trans. SE, **SE-10**, 5 (September 1984) 498–501.
- [51] R.G. Lanergan and B.A. Poynton, *Reusable code: The application development technique of the future*, Proceedings of the IBM SHARE/GUIDE Software Symposium, IBM, Monterey, CA, October 1979, 127–136.
- [52] D. Lea, *Christopher Alexander: An introduction for object-oriented designers*, Software Engineering Notes **19**, 1 (January 1994) 39–46.
- [53] Wayne C. Lim, *Effects of reuse on quality, productivity, and economics*, IEEE Software **11** (September 1994) 23–30.
- [54] M.A. Linton, J.M. Vlissides, P.R. Calder, *Composing user interfaces with Interviews*, IEEE Computer **22**, 2 (February 1989) 8–22.
- [55] Bertrand Meyer, *Reusability: The case for object-oriented design*, IEEE Software **4**, 2 (March 1987) 50–64.
- [56] Bertrand Meyer, *Lessons from the design of the Eiffel libraries*, CACM **33**, 3 (1991) 68–84.
- [57] Bertrand Meyer, *Applying Design by Contract*, IEEE Computer, 25(10), October 1992, 40–51.
- [58] Bertrand Meyer, **Reusable Software: The Base Object-Oriented Component Libraries**, Prentice-Hall International, 1994.
- [59] Mark Moriconi and Xiaolei Qian, *Correctness and composition of software architectures*, Proceedings of SIGSOFT’94: Foundations of Software Engineering, ACM Software Engineering Notes **19**, 5 (December 1994) 164–174.
- [60] D.L. Parnas, *On the criteria to be used in decomposing systems into modules*, Communications of the ACM **15**, 12 (1972) 1053–1058.

- [61] D.L. Parnas, *Designing software for ease of extension and contraction*, IEEE Trans. Software Eng. **SE-5**, 2 (1979) 128–137.
- [62] D.L. Parnas and P.C. Clements, *A rational design process: How and why to fake it*, IEEE Trans. Software Eng. **SE-12**, 2 (1986) 251–257.
- [63] C. Potts and G. Bruns, *Recording the reasons for design decisions*, Proceedings of the 10th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 418–427.
- [64] Jeff Poulin and Will Tracz, *WISR'93: 6th annual workshop on software reuse — Summary and workshop group reports*, ACM SIGSOFT Software Engineering Notes **19**, 1 (January 1994) 55–71.
- [65] R. Prieto-Díaz and J.M. Neighbors, *Module interconnection languages*, J. Systems and Software **6**, 4 (1987) 307–334.
- [66] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
- [67] Wilhelm Schäfer, Rubén Prieto-díaz, Masao Matsumoto, *Software Reusability*, Ellis Horwood, 1993.
- [68] G. Sindre, E.-A. Karlsson, T. Staalhane, *A method for software reuse through large component libraries*, **Proceedings ICCI'93**, Osman Abou-Rabia, Carl K. Chang, Waldemar W. Koczkodaj (editors), IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 464–468.
- [69] V. Singhal and D. Batory, *P++: A language for large-scale reusable software components*, Department of Computer Science, University of Texas at Austin, 1993.
- [70] Bjarne Stroustrup, **The C++ Programming Language**, 2nd edition Addison-Wesley, 1991.
- [71] M.E. Swanson and S.K. Curry, *Implementing an asset management program at GTE Data Services*, Information and Management **16** (1989).
- [72] Taligent, Inc., *Building object-oriented frameworks*, A Taligent White Paper, 1994.
- [73] W. Tracz, *The three cons of software reuse*, Proceedings of the Fourth Annual Workshop on Reuse, Syracuse, NY, 1991.
- [74] W. Tracz, *Domain analysis working group report*, First International Workshop on Software Reusability, Dortmund, Germany, July 3–5, 1991.
- [75] Panu Viljamaa, *The patterns busines: Impressions from PLoP-94*, ACM Software Eng. Notes **20**, 1 (January 1995) 74–78.
- [76] J.D. Warnier, **Logical Construction of Programs**, Van Nostrand, 1980.
- [77] A. Weinand, E. Gamma, R. Marty, *Design and implementation of ET++, a seamless object-oriented application framework*, Structured Programming **10**, 2 (1989) 63–87.
- [78] G. Wiederhold, P. Wegner, S. Ceri, *Towards megaprogramming*, CACM **35** (1992) 89–99.
- [79] Rebecca Wirfs-Brock and Ralph Johnson, *Surveying current research in object-oriented design*, CACM **33**, 9 (September 1990) 104–124.
- [80] Niklaus Wirth, *Program development by stepwise refinement*, Communications of the ACM **14**, 4 (1971) 221–227.
- [81] E. Yourdon and L.L. Constantine, **Structured Design**, Prentice-Hall, 1979.